INDEX

1. Introduction

Pneumonia is a respiratory infection that causes inflammation in the lungs, specifically the air-sacs known as Alveoli that get filled with pus. It is caused by bacteria, fungi and virus. The symptoms of this disease include coughing (with phlegm), nausea, fever, fatigue and breathlessness. This disease is communicable, i.e it can spread from one person to another via media like air and blood (at the time of birth)[1]. If left untreated, this condition can become lethal since it causes problems in the respiratory system of the person. People with a weak immune system, especially young children and the elderly, are more susceptible to Pneumonia. The infants may not show any signs of infection (visible symptoms) despite being very vulnerable to it, making the diagnosis of this disease in them quite difficult at the early stages.[2] As of 2019, Pneumonia accounted for 15% of overall deaths in children under the age of 5 and killed 7,40,180 children globally.[1]

As informed by the Health Minister of India in his response to a question raised during the question hour of the Rajya Sabha[3] (the upper house of the Parliament of India) on 5th April 2022, there was 1 doctor for every 854 persons in the country[4], which is roughly 1.17 doctors per 1000 persons. The number of available nurses in the country per 1000 people was 1.96[5],which is roughly 1 nurse for every 511 persons in the country. Even if the doctor-to-patient ratio had been better, the manual process of diagnosis still consumes a good amount of time.

Given the contagiousness of this disease, it has the potential to cause a mass-level epidemic the same way COVID-19 did. Such a pandemic-like situation may severely overwhelm the healthcare system , where it would neither be able to contain the spread of the disease nor treat as many patients. In such a time-critical situation, the process of manual diagnosis would require the resources (both human and non-human) that could be used in other areas. With the advent of technology, the diagnosis process can be automated to a certain degree. The computers are capable of performing the task much faster than the human brain. This would specially be helpful in the aforementioned situation where the diagnosis process has to be as quick as possible, while consuming least possible resources.

This project aims to design a Deep Learning model architecture that can automate the diagnosis process and provide the results as quickly as possible. It would be capable of detecting Pneumonia on the basis of the grayscale X-Ray image fed to it. The model prioritizes classifying Pneumonia X-Ray images correctly over everything else, since a false negative in this case would be detrimental. This asserts the fact that accuracy is not always the correct metric for evaluation. In this case, the *Recall* metric becomes a deciding factor. It is the fraction of samples classified correctly as positive, out of all the *actually positive* samples (and not all the samples as in the case of *Accuracy* metric).
At the same time, the model has to be as simple and less complex as possible. Simple models take less time to train and evaluate compared to the more complex ones. In order to simplify this, the model was designed using the appropriate number of *hyper-parameters* chosen using a probabilistic optimization technique called *Bayesian Optimization.*

Although tuning is an extra step, it greatly optimizes the time and the resources that would be required to develop a good model. However, the Bayesian Optimization method performs this task relatively quick. The time and resources required for the tuning process are still less than what would be required to repeatedly configure the model manually.

DATASET: https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia

1. WHO Fact Sheet on Pneumonia in Children: https://www.who.int/news-room/fact-sheets/detail/pneumonia
2. Mayo Clinic > Pneumonia > Symptoms and Causes: https://www.mayoclinic.org/diseases-conditions/pneumonia/symptoms-causes/syc-20354204
3. Ques: 353, List of Questions for ORAL ANSWERS [dated 05.04.2022], Rajya Sabha: https://cms.rajyasabha.nic.in/UploadedFiles/Questions/QuestionsList/256/English_202245_SQ05.pdf
4. PIB Press Release [ID: 1845081 dated 26.07.2022]: https://pib.gov.in/PressReleseDetailm.aspx?PRID=1845081
5. PIB Press Release [ID: 1813656 dated 05.04.2022]: https://pib.gov.in/PressReleseDetailm.aspx?PRID=1813656

2. Problem Statement

The task is to design a Deep Learning model architecture i.e. an Artificial Neural Network that is capable of distinguishing the X-Ray images of people with Pneumonia from those belonging to the normal people. This is a BINARY CLASSIFICATION problem; the X-Ray image fed to the neural network can either be classified as POSITIVE (i.e. WITH Pneumonia) or as NEGATIVE (i.e. WITHOUT Pneumonia).

Moreover, the priority is to detect as many *actually* positive samples as possible. Even if *some* negative images are wrongly classified as positive, it shall not be deciding factor in this case. This is due to the fact that the model serves as a *preliminary* screening tool whose task is to detect the samples that are subject to manual examination. It is NOT supposed to be a *perfect classifier* at the first place. Therefore, a high **Recall (True Positive Rate)** score is a prerequisite. However, it is not supposed to make so many *False Positive classifications* that it ends up drastically decreasing the **Precision** metric.

Although a very high precision is *not a prerequisite*, a very low precision would make the classifier classify more normal people as patients of Pneumonia that require further examination. Since they are not actually infected, the subsequent diagnosis would be redundant and eventually lead to wastage of time and resources.

Therefore, the classifier has to be designed considering both Precision and Recall scores. The score for the Recall metric has to be *very high* whereas even a *marginally high* score would be sufficient for the Precision metric. However, a high Precision score is always desirable, if not mandatory. To determine the combined effect of both Recall and Precision metrics, the metric **F-Score** is used. F-Score (also known as F1 Score) is the *harmonic mean* of the Recall and the Precision scores. Mathematically:

$$Recall = \frac{TP}{TP + FN} \qquad Precision = \frac{TP}{TP + FP} \qquad FScore = \frac{2.Precision.Recall}{Precision + Recall}$$

For checking the performance of the classifier at each iteration (epoch), the **Area Under the Curve (AUC) for ROC** can be used. ROC is the **Receiver Operating Characteristic** curve, a plot between TP and FP. The under this curve help in understanding how well the classifier is performing at different thresholds. A perfect classifier would have a AUC-ROC score of 1, whereas a totally bad classifier would have a AUC-ROC score of 0.

Combining all the aforementioned, the Problem Statement for the Project can be formulated as:

PROBLEM STATEMENT:
Design an Artificial Neural Network that performs a *Binary Classification* task on the X-Ray images of lungs, and classifies them as either NORMAL or PNEUMONIA. It should meet the following *minimum* benchmarks:

1. For **Training**:
   a) Binary Accuracy > 0.950
   b) Precision Score > 0.900
   c) Recall Score > 0.950
   d) F-Score > 0.925
   e) AUC-ROC Score > 0.850
2. For **Testing**:
   a) Binary Accuracy > 0.800
   b) Precision Score > 0.750
   c) Recall Score > 0.900
   d) F-Score > 0.820
   e) AUC-ROC Score > 0.800

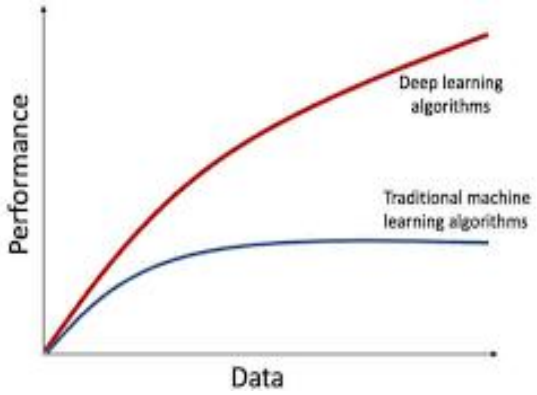
Try to construct a model as simple and optimized as possible and using minimum possible resources and time.

3. Theory

3.1: Limitations of Classical ML Techniques

Generally, the *classical* ML techniques are always preferable over Deep Learning, since the number of trainable parameters for ANNs are a lot more in number than those of classical ML models. Due to the high number of trainable parameters, the time and resource requirement for DL is much more than classical ML. Therefore DL requires *meticulous* planning before execution.

However, one clear advantage of DL over classical ML is that DL is capable of solving any AI-related problem, provided that the model has been appropriately designed. This becomes especially true when the data available for training the model is in excessive quantity. *The classical ML models' performance usually stagnates after a certain point. In comparison, DL models' performance improves with the trainable data, subject to the condition that the data has been properly transformed into a form that introduces new features to the model.* Therefore, data modelling is crucial in case of DL.



For this project, the reasons for selecting Deep Learning (DL) over other classical/traditional ML models were as follows:
1. The amount of data was large; the available number of images were in thousands.
2. The ANNs are capable of *learning* the localized features in the image much effectively than traditional ML models.
3. Classical ML techniques would have required more pre-training configurations, manual feature engineering and modelling in comparison to Deep Learning.
4. The images may not be easily separable into categories. The ML models work well when the degree of separation is known. E.g. Support Vector Machines (SVMs) work well on *linearly* separable data, but require explicit and complex data modelling to transform the raw data. Linear separability is a prerequisite for SVMs, but nor so much for DL.

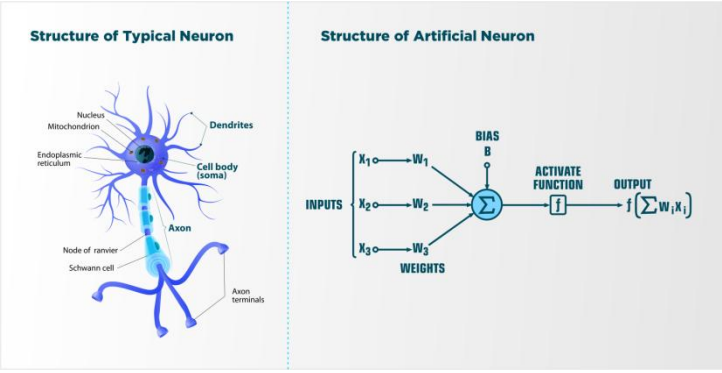
3.2: Deep Learning (DL)

Deep Learning is a *supervised* learning technique, i.e. the output of the training data is available for *supervising* the learning process. This is done by comparing the predicted outcome of an iteration against the actual outcome and transmitting the required changes back through the entire network. This helps in achieving better performance in the successive iterations by reducing the margin of error.

Although different from the classical/traditional form of ML, DL is still a sub-discipline of it, except that it specifically uses Artificial Neural Networks (ANNs) to perform the *learning* task. The fine line that separates *Classical* Machine Learning (ML) from Deep Learning (DL) is the fact that Classical ML methods are inspired from *statistical* processes whereas DL is inspired from *biological* processes. DL was developed to simulate the learning process of the human brain. The human brain functions by transmitting the signals among the inter-connected brain cells (neurons) of the nervous system. The ANNs perform the similar task of transmitting the input data through a network of inter-connected *Artificial Neurons*, also known as *Perceptrons*.

The neurons/perceptrons are the fundamental building blocks of the ANN. They are organized into layers stacked over each other. Every perceptron in one layer is connected to other neurons in the successive layers. These connections between the neurons create an entire network, which is known as the Artificial Neural Network (ANN).

3.2.1: Biological vs Artificial Neurons



Deep Learning was developed to simulate the working of a human brain to a certain degree, and perform the tasks in the same way as the human brain.
In the human brain, an individual cell receives multiple inputs from various sources (including other neurons) through its *Dendrites*. The signals received from all the sources are then transmitted through the *Axon*. Finally, they exit by the *Terminals* to other neurons/parts of the body. This process takes place for all the neurons in the nervous system. A well-coordinated concurrent execution of this process for multiple neurons is what forms the basis for brain's functioning.

Simulating this process, an artificial neuron receives the inputs from various sources. It then computes their *weighted sum*, adds a *Bias* term (a constant) and passes it to an *Activation Function*. The activation function then transforms this sum by mapping it from its current space to a different space. The final output from a neuron is then fed to the connected neurons in the successive layers. Mathematically, this process can be represented as:

$$x_i^{[l+1]} = f_l((W_i^{[l]})^T . X_i^{[l]} + b^{[l]}) \qquad f : \mathbb{R}^m \rightarrow \mathbb{R}^n$$

where:

- $x_i^{[l+1]}$ is the input for i[th] neuron in the next layer l+1.
- $X_i^{[l]}$ is the set of neurons in layer l that are connected to the neuron $x_i^{[l+1]}$ in the next layer l+1.
- $W_i^{[l]}$ is the set of weights such that weight $w_i^{[l]} \in W_i^{[l]}$ is multiplied with $x_i^{[l]} \in X_i^{[l]}$ while computing the weighted sum for the neuron $x_i^{[l+1]}$ in the next layer l+1. This matrix is transposed.
- $b^{[l]}$ is the bias term for the layer l.
- f_l is the activation function for the layer l that maps the weighted sum to the output of the neuron.

The aim of the entire training process is to determine the correct set of weights W which can be used to perform computations on the future data.
This is done by feeding the network with data repeatedly and computing the error by comparing the predicted output to the actual output. This is called **Feed-Forwarding**.
The computed error is then distributed back through the network by making adjustments to the weight. This is known as **Back Propagation**. Mathematically:

$$w_{i,new}^{[l]} = w_{i,old}^{[l]} - \epsilon \frac{\partial E}{\partial w_{i,old}^{[l]}} \qquad \forall \quad w_{i,old}^{[l]} \in W_j^{[l]} \subset W$$

where:

- $w_{i,old}^{[l]}$ and $w_{i,new}^{[l]}$ are the old and new values of i[th] weight in layer l respectively.
- E is the total error that is to be propagated back through the network.
- $W_j^{[l]}$ is the j[th] set of weights in layer l. W is the set of all the weights in the network.
- ϵ is the Learning Rate of the *Optimizer* used for finding the minimum Loss.

3.2.2: Loss Function

In order to compute the optimal weights for the neural network, there is a need to quantify the difference that exists between the model being trained and the required model. As this difference becomes small, the model would become better in terms of performance.

The function that quantifies how far the current state of trainable parameters is from the optimal state of the respective parameters (i.e. the target value for the trainable parameters) is known as the *Loss Function*. The aim of any model (classical ML model as well as DL model) is to reduce the loss as expressed by the loss function by taking the current state of trainable parameters as input. It would be a *representative* of how far the set of trainable parameters are from the optimal set of parameters (i.e. parameter values of the perfect model).

The type of loss function depends on the problem statement as well the model that is being used. For regression problems, the Mean Squared Error (MSE) and Mean Absolute Error (MAE) are used as loss functions. Minimizing these error terms help in determining the optimal values for the trainable parameters, which is the set of Regression Coefficients in this case.
For classification tasks, the loss function is in terms of the *Cross-Entropy*, a term that finds its origins in the Information Theory. Cross-entropy represents the difference between the probability distributions of different categories. For a multi-class classification problem, the *Categorical Cross-entropy* is used as the loss function. Mathematically, it can be defined as:

$$CE = -\sum_{r=1}^{N} p(y_r) log(\hat{y_r}) \qquad \forall \quad y_r \in \{0,1\} \;\; and \;\; \hat{y_r} \in [0,1]$$

where:

- y_r and $\hat{y}_r$ are the actual and predicted outcomes respectively.
- $p(y_r)$ is the probability of $y_r = 1$ for all values values of N.
- N is the number of categories.

3.2.2.1: Binary Cross-Entropy (BCE)

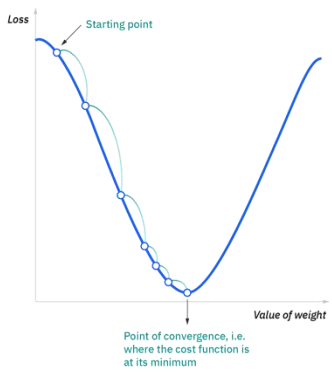
Since we are dealing with a *Binary Classification* problem, the value of n in this case would be 2. Also, the sum of probabilities of all the categories is always 1. Since there are only two categories, $p(y_1)$ and $p(y_2)$ can be written as p and 1-p. Hence, the Binary Cross-entropy can be expressed as:

$$BCE = -p(y_i) log(p(\hat{y_i})) - (1 - p_i) log(1 - p(\hat{y_i})) \qquad \forall \quad y_i \in \{0,1\} \;\; and \;\; \hat{y_i} \in [0,1]$$

where:

- $p(y_i)$ and $p(\hat{y_i})$ are the actual and predicted probabilities of $y_i = 1$.

3.2.3: Optimizer



In Machine Learning, an optimizer is a method/function that adjusts the trainable parameters after each cycle of feed-forwarding and back-propagation (also known as an epoch). In case of Deep Learning, the formula of the optimizer is used for updating the *weights* of the network after each epoch.
The most basic type of optimizer is the *Gradient Descent Optimizer*. The idea behind gradient descent algorithm is to begin with a large step size and keep decreasing it with respect to the distance left from the minima. The larger the distance, the smaller the step size. Note that this algorithm never provides the *exact* minimum value, but rather a value very close to it. The problem with Gradient Descent algorithm is that it may converge to the local minima and may not ever reach the global minimum. This is because its rate of descent decreases with the slope of the curve (difference in this case). The slope also

decreases at the local minima, giving a false impression that the global minimum is nearby. This results in incorrect predicted values and may lead to a very skewed accuracy rate. This can be mathematically represented as:

$$w_{i,t+1} = w_{i,t} - \eta(\frac{\partial L}{\partial w_i}) \qquad \forall\ w_{i,t} \in W$$

where $w_{i,t}$ represents the value of weight at time t, η is the Learning Rate of the optimizer, L represents the Loss Function for the problem and W is the set of all the weights.

3.2.3.1: Adaptive Moment Estimation (Adam) Optimizer

The Adam optimizer is designed to converge at the global minimum even in the presence of local minima. The Gradient Descent optimizer takes the value of the gradient at the moment of computation and keeps the learning rate constant for every weight update. Whereas Adam optimizer adjusts the learning rate for every individual weight update by considering the first as well as the second moment of the gradient. For a random variable *X*, the nth moment m_n can be written as: $m_n = \mathbb{E}(X^n)$.
Due to this property, the Adam optimizer is much faster and cost-effective in reaching the global minima in comparison to other optimizers.

3.2.4: Activation Functions

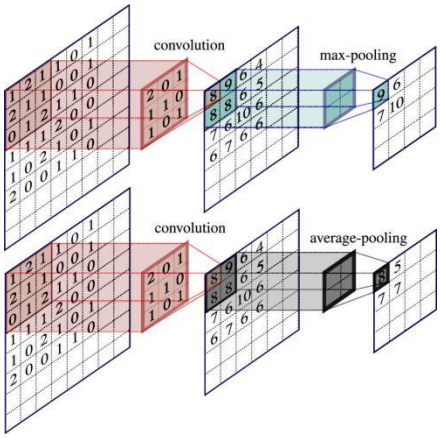
The aim of the activation functions is to transform the output of the weighted sum (as received by the neuron) from one space to another. Thereafter, this transformed output can either serve as an input for the neurons in the successive layers, or as the final output (in case the neuron belongs to the output layer). Some famous Activation functions are:

Name of the Activation Function	Formula f(x)	Characteristics of the Function
Linear	$f(x) = x$	No change in the value.
ReLU	$f(x) = max(x, 0)$	Capable of filtering out values less than a threshold value (in this case, 0).
Softmax	$f(x) = \frac{e^{-x}}{\sum_{r=1}^{n} e^{-r}}$	Can map any set of real numbers to a set of equivalent probabilistic values between 0 and 1; extensively used as the output function for Multi-class Classification problems.
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	Can confine any number within the the range [0,1]; extensively used as the output function for Binary Classification problems.
TanH	$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	Can map any real value to the range [-1,1].

In this project, we shall be using the ReLU and Sigmoid functions. We may use TanH as well, provided that this option is suggested while tuning the hyper-parameters for the final model.
ReLU function can be used in conjunction with the Convolutional layers. Given the filtering characteristic of this function, it can be used to isolate the main features from the noisy surroundings. This would provide specific features extracted from the image to the successive layers in the network. This would enable it to clearly learn from the features and not the noisy surroundings, thereby reducing the chances of model over-fitting the data.
Sigmoid function is used in the output layer of the ANN since the range of this function is [0,1]. This function can be used for classifying the X-Ray image as Normal/Pneumonic based on the value of the output. If the output is >0.5, then the image would be classified as 1 i.e. PNEUMONIA. Else it will be classified as 0, i.e. NORMAL.

3.2.5: Convolutions, ReLU Activation and Pooling

While feeding the images to an ANN, it is to be noted that the Image Classification task requires that the network learns the *common local features* from the images it is fed with. Therefore it becomes important to provide the network with *isolated local features*, which it may use on the future images. The manual process of extracting such features would be time-consuming as well as possibly very inaccurate.



Convolutional layers in the network are capable of extracting the localized features. This is done by performing mathematical convolutions on the feature space (in this case, the images in the matrix form). This can be mathematically represented as:

$$y_{a,b} = \sum_{j=1}^{n} \sum_{i=1}^{m} (k_{i,j}).(x_{a+i-1,b+j-1})$$

$$\forall \quad k_{i,j} \in K_{m*n} \quad x_{a,b} \in X_{M*N} \quad y_{a,b} \in Y_{M-\frac{m}{2},N-\frac{n}{2}}$$
$$\forall \quad a \in [1,m] \cap N \quad b \in [1,n] \cap N$$
$$\forall \quad K, X, Y \in \mathbb{R}^2$$

where:

- $k_{i,j}$ represents the element at the position (i,j) belonging to the Kernel space *K* of size m*n.
- $x_{i,j}$ represents the element at the position (i,j) belonging to the input space *X* of size M*N.
- $y_{i,j}$ represents the element at the belonging to the output space *Y* of size [M-(m/2)]*[N-(n/2)].

The resulting feature maps would then be filtered from the surrounding noise using the **ReLU function**. The ReLU function would filter out the negative values from the feature maps and output a new feature map containing the isolated feature.This can be mathematically represented as:

$$y_{new} = max(0, y_{old}) \qquad \forall \quad y_{old} \in Y \in \mathbb{R}^2$$

To further reduce the number of trainable parameters, the generated feature maps are reduced using the *Pooling* technique. Pooling is done by selecting a subset of the matrix and performing a specific operation over it. The results obtained by performing these operations over all the subsets of the original space provide the reduced feature space.
We have chosen **Max-Pooling** for our problem since this operation chooses the maximum value out of the subset of the the feature matrix. Since we want to be specific about the features and want them to be *highlighted*, we need to choose the maximum value in it to preserve the dominance of the feature in the final space.

3.2.6: Regularization

It may be possible that the model performs really well while training, but fails to give the similar outcome on the new data (such as test set). This condition is called *Over-fitting*, and is caused when the neural network learns more features from the training set than required. In DL, the *general* features are used to predict the outcome of the new data since these features are present in all samples (hence the term *general*). But the same extra features may not be present in the new samples.
In order to overcome this issue, the model needs to be made less complex. This is achieved by *penalizing* the model for being too complex by incorporating structural changes to the model. These changes are made in such a way that the overall decrease in loss does not bring equivalent change in the model as it would have in the absence of Regularization. In other words, only a fraction of the total information gained during the learning process is used for changing the trainable parameters.

3.2.6.1: Dropout Regularization

Considering the fact that the neurons present in the network are the basic units involved in the learning process, we can alter the amount of information learned by the network by changing the number of

neurons and their configuration in the network. Over-fitting occurs when the information learned from the training set is not general but also more than what is required.
In Dropout Regularization, a fraction of the neurons present in a particular layer are temporarily *dropped out*, i.e. they don't participate in the learning process like the other fraction (that learns the information at that moment). These neurons are selected randomly. In this way, the neurons learn using limited information, thereby preventing the chances of over-fitting the data. Each neuron in such a layer would therefore contain less information than it would have contained in the absence of Dropout Regularization.

3.2.6.2: Batch Normalization

Normalization is the process of confining a set of values to the range [0,1] by using the mean and the standard deviation about the mean value. This is achieved by using the formula:

$$x_{new} = \frac{x_{old} - \mu}{\sigma} \qquad \forall \quad x_{old} \in \mathbb{R} \qquad x_{new} \in [0, 1]$$

where: μ and σ are the Mean and the Standard Deviation for the set *X* respectively.

When the data is fed to the network in batches, the values of the inputs may change drastically and the gradients of the vector space X may *vanish/explode*. This is very likely to happen in case of Convolutional Networks, whose output feature space has a large range.
When the normalization is applied to the input data in the batch, the values are normalized using the mean and the standard deviation of the batch data. This is know as *Batch Normalization*.
This helps in preventing over-fitting by confining the information filtered by the convolutions to the range [0,1]. This even makes the computations easier and faster since the values now lie in a *Standard Normal Distribution with fixed characteristics*.

3.2.7: Hyper-parameters

In ML/DL, parameters are the entities that are determined *during* the training process. However, to correctly determine the parameters, the initial configuration of the model needs to be set appropriately. **Hyper-parameters** are those entities which are determined and fixed *before* training the model on the training data. They are not influenced by the training process. The values for the hyper-parameters are a major factor in determining the optimal values for the trainable parameters.
The examples of Hyper-parameters in Deep Learning include the number of hidden layers, number of neurons in each layer, the number of filters in a Convolution layer, the learning rate of the optimizer, etc. Note that these have to be determined prior to training the neural network. It is *after* determining these *hyper-parameters* that the model can be trained on the training dataset to determine the trainable *parameters* i.e. the weights between the neurons.

3.2.7.1: Hyper-parameter Tuning

Hyper-parameter Tuning is the process of determining the *optimal* set of hyper-parameters *before the training process*, that shall produce the optimal values for the trainable parameters *after the training process*. There are multiple processes to perform Hyper-parameter Tuning.

Grid Search is a *brute-force method* to try out *every* possible combination of hyper-parameters to evaluate the metric, and choose the set that generates the maximum or the minimum value (depending on the type of the evaluation metric). The downsides of this process is that it is computationally very expensive and works well only if the hyper-parameters are in small finite numbers.

Another method to perform the task is **Random Search**. This technique *randomly* picks a finite number of hyper-parameter combinations and chooses the best set among them. Basically, it performs Grid Search on a *subset* of all hyper-parameter combinations rather that the *entire* set of combinations. Although it reduces the time and computational resources required by the Grid Search method, it does not *guarantee* the optimal combination of hyper-parameters required for the problem. This is because the combinations are randomly sampled, which means that there is a chance that the actual optimal combination may not be a part of the random sample.

The technique that we have used in this project for tuning the hyper-parameters is **Bayesian Optimization**. This technique leverages the *computational advantage of Random Search* and *the guarantee of optimal value as in Grid Search*, while using a *probabilistic* approach to determine the optimal combination.

3.3 Bayesian Optimization

Bayesian Optimization is an optimization technique that determines the optimal value of an input (or a set of inputs) that shall maximize or minimize (depending on the objective) the output of an *unknown* function from a set of *given/generated* outputs. The process applies Bayesian method to a Gaussian distribution to perform the task. Note that it determines the optimal input values that generates the optimal output *without determining the actual function* that maps those input values to their corresponding outputs.

There are three elements of this technique:

- **Black-Box function**: The *unknown* function that maps the input values to the output values.
- **Surrogate Function**: The function that is constructed by repeatedly appending the chosen values of input-output sets after each iteration. As the set grows and tends towards the global set, the surrogate function becomes closer to the unknown Black-Box function. This can be mathematically expressed as:

  $$B(D) = \lim_{Y_t \to D(y)} S(Y_t) \qquad \forall \quad Y_t \subseteq D(y)$$

  where:
  - Y_t is the subset of output values sampled from y component (i.e. the output component), which is the subset of the global dataset D
  - $B(D)$ is the unknown Black-Box function generated using the y component (i.e. the output component) of the global dataset D.
  - $S(Y_t)$ is the Surrogate function formed using the sampled subset of outputs: Y_t .

  The above expression says that as the subset Y_t becomes large and proceeds towards becoming $D(y)$, the surrogate function $S(x)$ starts behaving in the same way as the unknown black-box function $B(D)$.

- **Acquisition Function**: The most crucial element of this technique that picks the next input value to be tested. This value would improve the current state and lead us closer to finding the unknown black-box function. For this particular case, it would help in finding the global maxima/minima for the unknown Black-Box function *B(x) without determining the function itself*.

The technique uses a small sample from the search space to begin with, containing both inputs and the outputs. This subset is then *explored* and the corresponding outputs construct the initial surrogate function using interpolation. The acquisition function then cautiously chooses the next input from the *unexplored set* based on the improvement that the chosen input is *expected* to generate.

After choosing the input-output set, it then appends this *chosen* set to the *sample*, updates the surrogate function, transforms itself after updating the surrogate function, and repeats the operation by again choosing the next input value from the *unexplored set of inputs* in the similar way. This keeps repeating until the optimal input set is found, or the number of maximum iterations is reached.

3.3.1: Acquisition Function

The Acquisition Function is the most crucial element for Bayesian Optimization as it fetches the next input set to be tested. For this particular problem, the acquisition function can be mathematically expressed using the following set of equations:

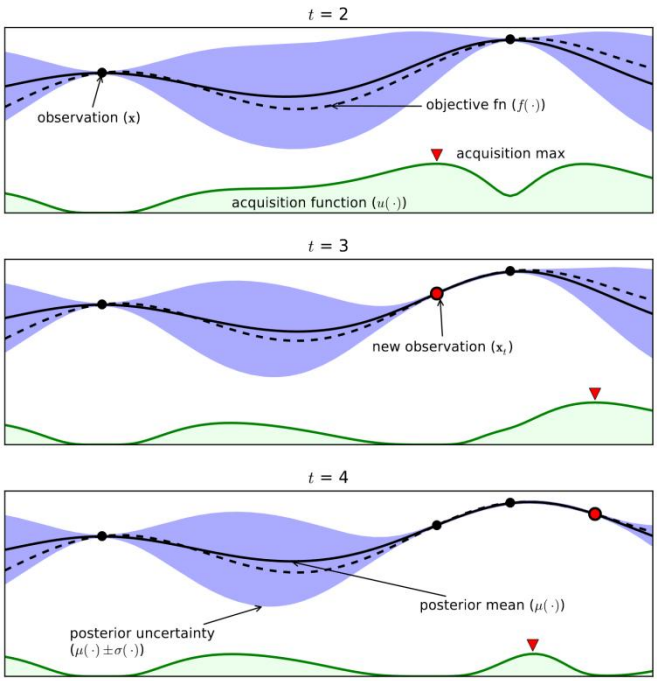
$$y_{best} = max(Y_t) \qquad x_{best} = argmax(Y_t)$$

$$UCB(Y_t) = \mu(Y_t) + k\sigma(Y_t)$$

$$S : max(UCB(Y_t), y_{best})$$

$$x_{t+1} = argmax(S)$$

$$\forall \quad k \in \mathbb{R}^+, \;\; x_{t+1} \subseteq D(x) \in D, \;\; Y_t \subseteq D(y) \in D$$



In the equations, UCB is the *Upper-Confidence Bound of the confidence intervals* i.e. the function formed using the *upper outline* of the blue region in the figure.

Y_t is the set of sampled outputs i.e. the black points in the figure. It is the subset of all the *outputs* sampled so far and belong to the global search space *D*.
These points construct and update the Surrogate Function *S* i.e. the solid black line in the figure, described in the figure as *Posterior Uncertainty*.

The Acquisition Function (green solid line at the bottom of each sub-figure) is constructed by taking the UCB i.e. the upper outline of the blue region *that lies above the maximum value in* Y_t.

The *new point* to be checked is picked up using the Acquisition Function. The point to be explored is the one that generates the maximum value in the acquisition function. The actual output at this *input* point is compared to the best output value found so far i.e. y_{best}.
If the observed value is greater than y_{best}, then the fetched point becomes the new optimal argument, i.e. x_{best}, and the output of this point becomes the new y_{best}. The Surrogate function is transformed accordingly after incorporating the newly fetched point.
This process is repeated until the best input is found i.e. the input that generates the maximum value among the given outputs. The process ends when either the Acquisition Function flattens i.e. the search space gets fully explored, or the iteration limit is reached. The end value of x_{best} is the required output of the whole process.

Note that we try to find the global maxima without figuring out the actual Black-Box function i.e. the dotted line in the figure.

3.3.2: Using Bayesian Optimization in Hyper-parameter Tuning

The aforementioned process is executed for tuning the hyper-parameters *before* training the model on the training set. This can be analogized in the following way:

- The x_{t+1} is the combination of hyper-parameter values to be explored in the next trial.
- y_{best} is the maximum value of the evaluation metric *Binary Accuracy* that has been determined till the current trial.
- Y_t is the set of binary accuracy explored so far.
- x_{best} is the best combination of hyper-parameter values found so far, the one that generates y_{best}.
- The process continues for a fixed number of *trials*, or until the best combination of hyper-parameter values is confirmed.
- The end value of x_{best} would be the result of the Hyper-parameter Tuning Process.

4. Project Implementation

4.1: Implementation Tools

Tool	Purpose	Speciality
Kaggle	Source of data	● Large repository of structured data. ● The dataset is usually well-organized and well-defined. ● Good Community support in case of problem.
Google Colab	Project Environment	● Can implement both coding as well as markdown blocks ● Provides high processing support (NVIDIA Tesla K80 GPU) support (but with certain limitations). ● Large collection of pre-installed libraries for Python programming. ● Works well with *Package Installer for Python (PIP)*. for downloading new packages. ● Provide built-in functionality to execute Linux commands. ● *Intellisense* for auto-completing the code.
TensorFlow/Keras	For implementing Deep Learning process	● Keras library provides a very convenient way to implement DL models. ● TensorFlow framework provides great backend support while executing the entire process irrespective of the environment. Keras runs as a high-level API on top of TensorFlow backend. ● Provides access to very high processing resource i.e. *Tensor Processing Unit (TPU)* if required. ● Great community support. ● Provides good control over the DL training process using *Callbacks*. ● Provides Image Pre-processing and Augmentation capability *at runtime* using generators.
TensorBoard	DL Process Visualization	● Automatically updates after each refresh. ● Real-time visualization of training process. ● Integrable with the Google Colab environment. ● No sophisticated coding required. ● Results can be stored and re-produced.
Keras-Tuner	Hyper-parameter Tuning	● Provides a variety of optimization methods. ● Can implement Bayesian Optimization on a Keras DL model. ● Totally compatible with TensorFlow/Keras. ● Results can be stored and re-produced. ●
Python	Programming Language	● Easy syntax with mandatory indentation. ● Great collection of libraries (>60,000) with excellent community support. ● Very convenient for Data Analysis (using NumPy and Pandas libraries) and Data Visualization (using MatPlotLib and Seaborn). ● Established practices and standards as described under the *Python Enhancement Program (PEP)*.

4.2: Methodology for Project Implementation

The following steps were followed while implementing the project:

1. The process of implementation was planned and the evaluation metrics were determined.
   Reference: Section 2 - Problem Statement.

2. The dataset was downloaded from **Kaggle** into the **Google Colab** environment in the form of a ZIP file. After unzipping, it was found that the dataset was already segmented into Training, Validation and Test set in appropriate proportion (visualized on the Kaggle platform).
   Reference: https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia

3. The dataset was analyzed and interpreted in line with the problem statement. Thereafter, the tools for implementing the project were determined.
   Reference: Section 4.1 - Tools for Implementation.

4. The required libraries were figured out, installed using PIP (only Keras-Tuner) and imported into the Google Colab environment.
   Reference: Section 4.5 - Code Implementation.

5. The variables, constants, methods and classes to be used in global space were defined:
   a) **Variables**: addresses for the dataset and other file-paths.
   b) **Constants**: batch size, target image size and the list of metrics for evaluation.
   c) **Methods**:
      i. Plotting the *Confusion Matrix* for the obtained results.
      ii. Displaying the obtained values of metrics after testing.
      iii. Transforming the the data into trainable form using the ***ImageDataGenerator class in TensorFlow/Keras***. This class has the following speciality:
         - Fetching data *directly* from the directories structured in a certain way.
         - Performing *Feature Scaling* and *Image Augmentation* at the runtime.
         - Returning a generator, which is faster and occupies less memory than Python lists.
   d) **Classes**: custom callbacks by inheriting the ***Callbacks class in TensorFlow/Keras*** to halt the training process in the middle of execution in case of:
      i. **Stagnation** i.e. no improvement in performance after a certain number of epochs.
      ii. **Early Achievement** i.e. achieving the required target before reaching the epoch limit.
   Reference: Section 4.5 - Code Implementation.

6. *Exploratory Data Analysis (EDA)* was performed on the dataset using the following method:
   a) Distribution of the Dataset - Type and Categories.
   b) Size and dimension of images.
   c) Plotting the sample images from each category in the training set.
   d) Plotting the Mean Image of the images in the training set.
   e) Plotting the Mean Absolute Difference-Mean Fraction Image from the images in the training set.
   Reference: Section 4.3.2 - Exploratory Data Analysis (EDA).

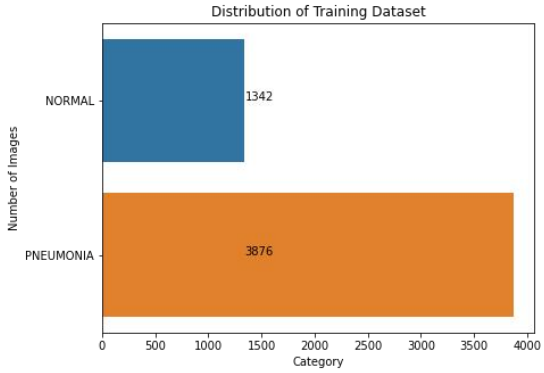
7. A check model was implemented to see the working of the most basic model and its results were analyzed along with those of EDA to design the actual model's architecture. The results were visualized using the *TensorBoard* visualization tool offered by TensorFlow.
   Reference: Section 4.5 - Code Implementation.

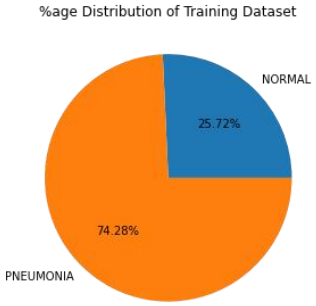
8. The hyper-parameters were identified and their search spaces were determined. Only those hyper-parameter values were calculated through the tuning process which could not be figured out on a theoretical basis. The final model's architecture was determined using the values obtained from the tuning and miscellaneous .
   Reference: Section 4.3.3 - Model Design.

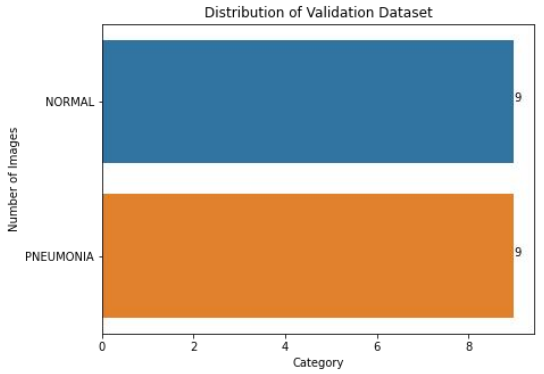
9. The final model was trained on the training set and evaluated over the test set. The obtained results were analyzed by comparing them with the criteria mentioned in the problem statement.
   Reference: Section 5 - Result Analysis.

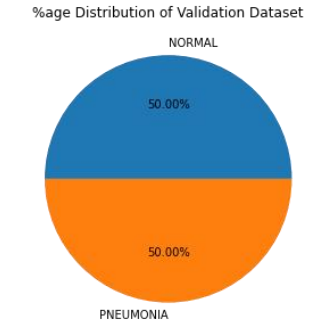
4.3: Analyzing the Implementation Metadata

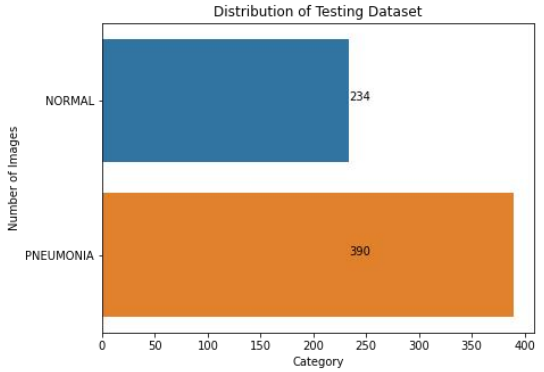
4.3.1: Distribution of the Dataset.

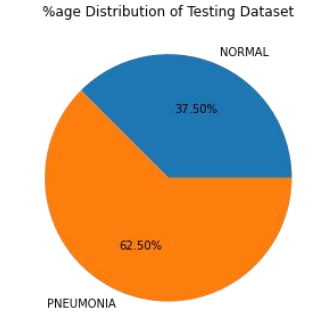












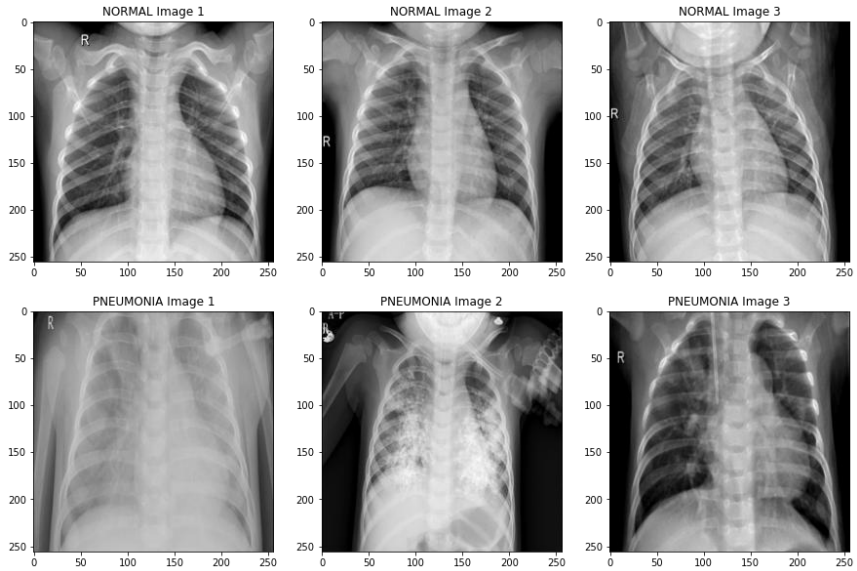
4.3.2: Exploratory Data Analysis (EDA)

4.3.2.1: Size and Dimensions of the Images

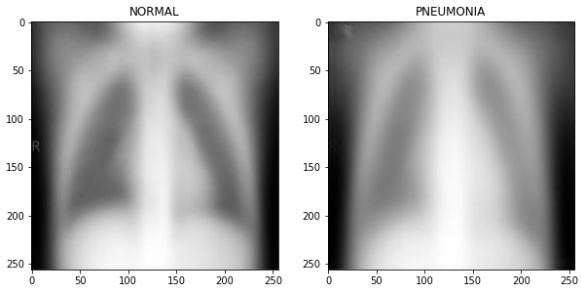
```
Min Width:   127
Max Width:   2663
Min Height:   384
Max Height:   2916
Avg Width:   1005.840357306459
Avg Height:   1354.620705451214
```

It was found that the size and the dimensions of the images were different. However, it did not require any explicit pre-processing since *the* ***ImageDataGenerator*** *class in TensorFlow/Keras re-orients the image to a target configuration* using the target_size argument. In this case, that target configuration was (256,256,3).

4.3.2.2: Sample Images (by Category)

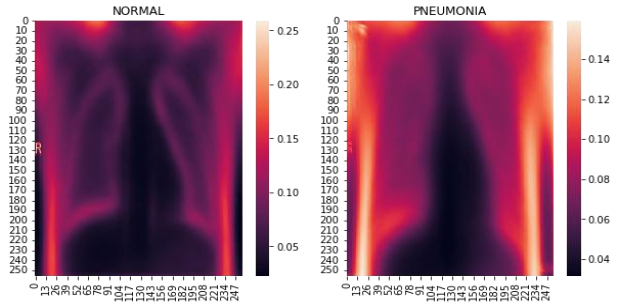


4.3.2.3: Mean Images (by Category)



As visible in the mean images of both the categories, there seems to be little variation in the position-specific features. However, it is observable that the mean image for NORMAL images and that of PNEUMONIA images differ at the lungs region over a good amount of area (PNEUMONIA images are lighter in the lung region). Therefore the extent of *breaking down* the image into components is relatively low. However, we would still need to see the pixel-wise variation about the mean image to arrive at a conclusive judgement.

4.3.2.3: M.A.D.-Mean Fraction Images (by Category)



As visible in the adjacent image, the pixel-wise Mean Absolute Deviation (M.A.D.) about the Mean Image in both the categories is well within 10% in the lungs region. However, there still exists a difference of 2-3% in the plot for NORMAL and PNEUMONIA image dataset. Considering this fact and *after running the check model*, we can assume that 2 Convolutional layers with ReLU activation and a MaxPooling layer each should be sufficient for segmenting the features of the image in a way that the Dense block can perform the classification operation on.

To extract maximum features using the minimum number of filters, we can have a lower number of filters for the first Convolutional layer (4, 8 or 16) in comparison to those in the second Convolutional layer (16, 32 or 64). The first Convolutional Layer would filter out the high-level features, and the second layer would filter out the lower-level features from the feature maps produced by the first Convolutional layer. The ReLU activation in both the layers would filter out the noise surrounding the feature, and the max-pooling operation would emboss the feature in comparison to the surroundings. This would also reduce the dimensions of the image by the time it reaches the dense block.

4.4: Model Design

4.4.1: Optimal Hyper-parameters

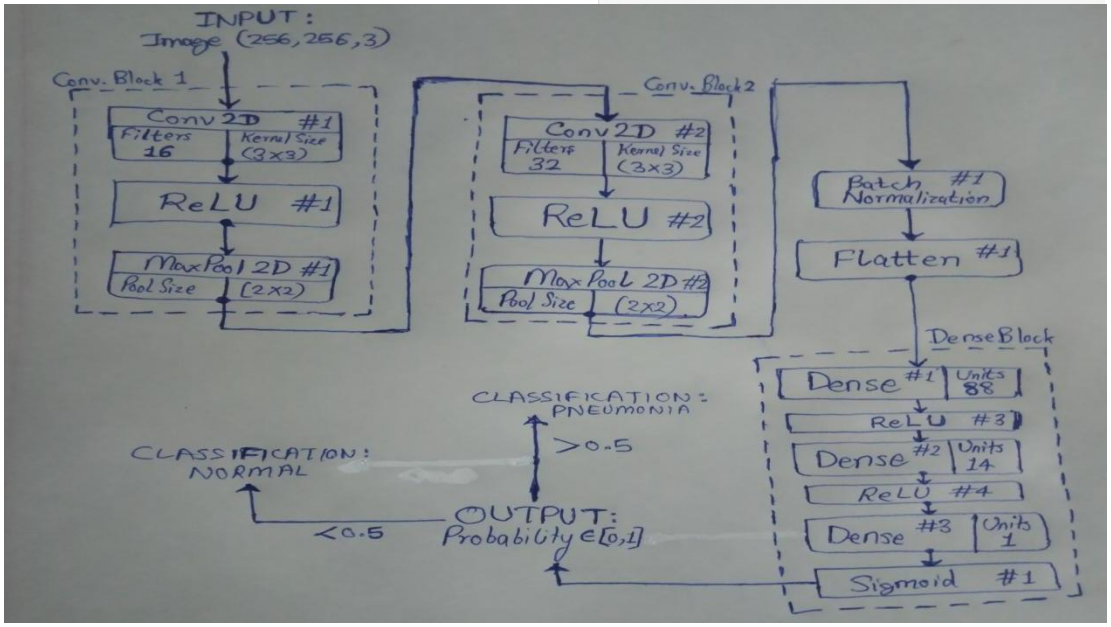
	Hyperparameter	Optimal Value
0	Filters_1	16
1	Filters_2	32
2	Units_1	88
3	Activation_1	relu
4	Dropout_Rate_1	0.233503
5	Units_2	14
6	Activation_2	relu
7	Learning_Rate	0.000118

4.4.2: Model Architecture

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 16)      448

 max_pooling2d (MaxPooling2D  (None, 127, 127, 16)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 32)      4640

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 32)       0
 2D)

 batch_normalization (BatchN  (None, 62, 62, 32)       128
 ormalization)

 flatten (Flatten)           (None, 123008)            0

 dense (Dense)               (None, 88)                10824792

 dropout (Dropout)           (None, 88)                0

 dense_1 (Dense)             (None, 14)                1246

 dense_2 (Dense)             (None, 1)                 15

=================================================================
Total params: 10,831,269
Trainable params: 10,831,205
Non-trainable params: 64
_________________________________________________________________
```

```python
final_model = tf.keras.Sequential(
    layers = [
        tf.keras.layers.Conv2D(
            filters = 16,
            kernel_size = (3,3),
            activation="relu",
            input_shape=IMG_SHAPE
        ),
        tf.keras.layers.MaxPool2D((2,2)),
        tf.keras.layers.Conv2D(
            filters = 32,
            kernel_size = (3,3),
            activation = "relu"
        ),
        tf.keras.layers.MaxPool2D((2,2)),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(
            units = 88,
            activation = "relu"
        ),
        tf.keras.layers.Dropout(0.233503),
        tf.keras.layers.Dense(
            units = 14,
            activation = "relu"
        ),
        tf.keras.layers.Dense(1, activation="sigmoid")
    ]
)
```

```python
# Binding the model with Algorithmic parameters:
final_model.compile(
    optimizer = tf.keras.optimizers.Adam(
        learning_rate = 0.000118
    ),
    loss = tf.keras.losses.BinaryCrossentropy(),
    metrics = METRICS
)
```



4.5: Code Implementation

PNEUMONIA CLASSIFICATION

1: Project Setup

Before downloading the dataset, we need to upload the **Kaggle.json** file for downloading the dataset from Kaggle

1.1: Getting the dataset from Kaggle

```
In [ ]: ! pip install kaggle
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: kaggle in /usr/local/lib/python3.8/dist-packages (1.5.12)
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/dist-packages (from kaggle) (4.64.1)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.8/dist-packages (from kaggle) (1.15.0)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-packages (from kaggle) (2.23.0)
Requirement already satisfied: certifi in /usr/local/lib/python3.8/dist-packages (from kaggle) (2022.9.24)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.8/dist-packages (from kaggle) (2.8.2)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.8/dist-packages (from kaggle) (7.0.0)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.8/dist-packages (from kaggle) (1.24.3)
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/lib/python3.8/dist-packages (from python-slugi
fy->kaggle) (1.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests->kaggle)
(2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.8/dist-packages (from requests->kagg
le) (3.0.4)
```

```
In [ ]: ! mkdir ~/.kaggle
```

```
In [ ]: ! cp kaggle.json ~/.kaggle
```

```
In [ ]: ! chmod 600 ~/.kaggle/kaggle.json
```

```
In [ ]: ! kaggle datasets download paultimothymooney/chest-xray-pneumonia
```

```
Downloading chest-xray-pneumonia.zip to /content
100% 2.29G/2.29G [01:46<00:00, 23.0MB/s]
100% 2.29G/2.29G [01:46<00:00, 23.1MB/s]
```

1.2: Unzipping the Dataset

```
In [ ]: ! unzip chest-xray-pneumonia.zip
```

1.3: Libraries

```
In [ ]: ! pip install keras-tuner
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting keras-tuner
  Downloading keras_tuner-1.1.3-py3-none-any.whl (135 kB)
     |████████████████████████████████| 135 kB 37.3 MB/s
Requirement already satisfied: ipython in /usr/local/lib/python3.8/dist-packages (from keras-tuner) (7.9.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-packages (from keras-tuner) (21.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from keras-tuner) (1.21.6)
Requirement already satisfied: tensorboard in /usr/local/lib/python3.8/dist-packages (from keras-tuner) (2.9.1)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-packages (from keras-tuner) (2.23.0)
Collecting kt-legacy
  Downloading kt_legacy-1.0.4-py3-none-any.whl (9.6 kB)
Requirement already satisfied: prompt-toolkit<2.1.0,>=2.0.0 in /usr/local/lib/python3.8/dist-packages (from ipy
thon->keras-tuner) (2.0.10)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tune
r) (0.7.5)
Requirement already satisfied: pexpect in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tuner)
(4.8.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.8/dist-packages (from ipython->keras-
tuner) (57.4.0)
Requirement already satisfied: decorator in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tuner)
(4.4.2)
Collecting jedi>=0.10
  Downloading jedi-0.18.2-py2.py3-none-any.whl (1.6 MB)
     |████████████████████████████████| 1.6 MB 52.6 MB/s
Requirement already satisfied: backcall in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tuner)
(0.2.0)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tu
ner) (5.6.0)
Requirement already satisfied: pygments in /usr/local/lib/python3.8/dist-packages (from ipython->keras-tuner)
(2.6.1)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /usr/local/lib/python3.8/dist-packages (from jedi>=0.10->
ipython->keras-tuner) (0.8.3)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.8/dist-packages (from prompt-toolkit<2.1.0,>=
2.0.0->ipython->keras-tuner) (0.2.5)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.8/dist-packages (from prompt-toolkit<2.1.0,
>=2.0.0->ipython->keras-tuner) (1.15.0)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist-packages (from packagi
ng->keras-tuner) (3.0.9)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.8/dist-packages (from pexpect->ipython
->keras-tuner) (0.7.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.8/dist-package
s (from requests->keras-tuner) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests->keras-tun
er) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.8/dist-packages (from requests->kera
s-tuner) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests->ker
as-tuner) (2022.9.24)
Requirement already satisfied: absl-py>=0.4 in /usr/local/lib/python3.8/dist-packages (from tensorboard->keras-
tuner) (1.3.0)
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from tensorboard->ker
as-tuner) (1.0.1)
Requirement already satisfied: protobuf<3.20,>=3.9.2 in /usr/local/lib/python3.8/dist-packages (from tensorboar
d->keras-tuner) (3.19.6)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/local/lib/python3.8/dist-packages
(from tensorboard->keras-tuner) (0.6.1)
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.8/dist-packages (from tensorboard->keras-t
uner) (0.38.4)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.8/dist-packages (from tensorboard->ker
as-tuner) (3.4.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.8/dist-packages (from tensorboar
d->keras-tuner) (2.15.0)
Requirement already satisfied: grpcio>=1.24.3 in /usr/local/lib/python3.8/dist-packages (from tensorboard->kera
s-tuner) (1.51.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.8/dist-packages (from te
nsorboard->keras-tuner) (1.8.1)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.8/dist-packages (from
tensorboard->keras-tuner) (0.4.6)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.8/dist-packages (from google-au
th<3,>=1.6.3->tensorboard->keras-tuner) (5.2.0)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.8/dist-packages (from google-aut
h<3,>=1.6.3->tensorboard->keras-tuner) (0.2.8)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.8/dist-packages (from google-auth<3,>=1.
6.3->tensorboard->keras-tuner) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.8/dist-packages (from google-
auth-oauthlib<0.5,>=0.4.1->tensorboard->keras-tuner) (1.3.1)
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.8/dist-packages (from markdown
>=2.6.8->tensorboard->keras-tuner) (4.13.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.8/dist-packages (from importlib-metadata>=4.
4->markdown>=2.6.8->tensorboard->keras-tuner) (3.11.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.8/dist-packages (from pyasn1-modu
les>=0.2.1->google-auth<3,>=1.6.3->tensorboard->keras-tuner) (0.4.8)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.8/dist-packages (from requests-oauthli
b>=0.7.0->google-auth-oauthlib<0.5,>=0.4.1->tensorboard->keras-tuner) (3.2.2)
Installing collected packages: jedi, kt-legacy, keras-tuner
Successfully installed jedi-0.18.2 keras-tuner-1.1.3 kt-legacy-1.0.4
```

```
In [ ]: import warnings
warnings.filterwarnings("ignore")
```

```
In [ ]: import os
import PIL
import random
import zipfile
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
import keras_tuner as kt

from matplotlib import pyplot as plt
```

```
In [ ]: library = {
    "Keras-Tuner": kt,
    "MatPlotLib": matplotlib,
    "NumPy": np,
    "Pandas": pd,
    "Pillow": PIL,
    "Seaborn": sns,
    "TensorFlow": tf
}
pd.DataFrame(
    data = {
        "Library": [ x for x in library.keys() ],
        "Version": [ x.__version__ for x in library.values() ]
    }
).set_index("Library")
```

Out[ ]:

Library	Version
Keras-Tuner	1.1.3
MatPlotLib	3.2.2
NumPy	1.21.6
Pandas	1.3.5
Pillow	7.1.2
Seaborn	0.11.2
TensorFlow	2.9.2

2: Global Space

2.1: Directory Variables

2.1.1: Data Source

```
In [ ]: TRAIN_DIR = "chest_xray/chest_xray/train"
VALID_DIR = "chest_xray/chest_xray/val"
TEST_DIR = "chest_xray/chest_xray/test"
```

2.1.2: TensorBoard Paths

```
In [ ]: TB_CHECK_LOGS = "TB_Logs/CheckModel"
TB_TUNE_LOGS = "TB_Logs/TuningModel"
TB_FINAL_LOGS = "TB_Logs/FinalModel"
```

2.2: Constants

```
In [ ]: BATCH_SIZE = 8
IMG_SHAPE = (256,256,3)
```

```
In [ ]: METRICS: list = [
    tf.keras.metrics.BinaryAccuracy(),
    tf.keras.metrics.TruePositives(),
    tf.keras.metrics.TrueNegatives(),
    tf.keras.metrics.FalsePositives(),
    tf.keras.metrics.FalseNegatives(),
    tf.keras.metrics.AUC(curve="ROC")
]
```

2.3: Methods

2.3.1: Plotting the Confusion Matrix

```
In [ ]: def plot_confusion_matrix(TP: int, FP: int, TN: int, FN: int):
    """
    Objective:
        To plot the confusion matrix for the given number
        of True/False Positives/Negatives.

    Args:
        TP  : the number of True Positives.
        FP  : the number of False Positives.
        TN  : the number of True Negatives.
        FN  : the number of False Negatives.

    Returns : None
    """

    arr = np.array([ TP, FP, FN, TN ])
    matrix = np.reshape(
        np.array(arr*100/sum(arr)),
        newshape = (2,2)
    )
    _,ax = plt.subplots()
    for i in range(2):
        for j in range(2):
            text = str(round(matrix[i][j],2))+"%"
            ax.text( i, j, text, va="center", ha="center" )
    ax.matshow( matrix, cmap="Blues" )
    plt.xticks( ticks=[0,1], labels=["PNEUMONIA","NORMAL"] )
    plt.xlabel("Actual")
    plt.yticks( ticks=[0,1], labels=["PNEUMONIA","NORMAL"] )
    plt.ylabel("Predicted")
    plt.show()
```

2.3.2: Displaying the Metrics

```
In [ ]: def display_metrics(TP: int, FP: int, TN: int, FN: int):
    """
    Objective:
        To plot the confusion matrix for the given number
        of True/False Positives/Negatives.

    Args:
        TP  : the number of True Positives.
        FP  : the number of False Positives.
        TN  : the number of True Negatives.
        FN  : the number of False Negatives.

    Returns : None
    """

    metrics = {
        "Metric": [
            "True Positives", "True Negatives",
            "False Positives", "False Negatives",
            "Binary Accuracy", "Precision",
            "Recall/Sensitivity","Specificity",
            "F-Score"
        ],
        "Value": [
            TP,TN,
            FP,FN,
            (TP+TN)/(TP+TN+FP+FN), TP/(TP+FP),
            TP/(TP+FN), TN/(TN+FP),
            (2*TP)/(2*TP+FN+FP)
        ]
    }

    print("METRICS OF EVALUATION")
    print(pd.DataFrame(metrics).set_index("Metric"))
```

2.4: Classes

2.4.1: Custom Callbacks

```
In [ ]: class EarlyStopAtStagnancy(tf.keras.callbacks.Callback):

    def __init__( self, patience ):
        super(EarlyStopAtStagnancy,self).__init__()
        self.patience = patience
        self.best_weights = None

    def on_train_begin( self, logs=None ):
        self.wait = 0
        self.stop_epoch = 0
        self.best_epoch = 0
        self.best_acc = 0
        self.best_val_acc = 0

    def on_epoch_end( self, epoch, logs=None ):
        acc_at_epoch = logs.get("binary_accuracy")
        val_acc_at_epoch = logs.get("val_binary_accuracy")
        if (acc_at_epoch < self.best_acc
                or val_acc_at_epoch < self.best_val_acc
            ):
            self.wait += 1
            if self.wait == self.patience:
                self.stop_epoch = epoch
                self.model.stop_training = True
                self.model.set_weights(self.best_weights)
        else:
            self.best_acc = acc_at_epoch
            self.best_val_acc = val_acc_at_epoch
            self.wait = 0
            self.best_weights = self.model.get_weights()
            self.best_epoch = epoch

    def on_train_end( self, logs=None ):
        if self.stop_epoch > 0:
            print("--- TRAINING STOPPED AT EPOCH: ",
                  self.stop_epoch, " DUE TO STAGNANCY ---")
            print("--- RESTORING THE WEIGHTS FROM EPOCH: ",
                  self.best_epoch, " ---")
```

```
In [ ]: class EarlyStopAtBestMetrics(tf.keras.callbacks.Callback):

    def __init__(self):
        super(EarlyStopAtBestMetrics,self).__init__()
        self.best_weights = None

    def on_train_begin( self, logs=None ):
        self.stop_epoch = 0
        self.best_train_acc = 0
        self.best_val_acc = 0

    def on_epoch_end( self, epoch, logs=None ):
        train_acc_at_epoch = logs.get("binary_accuracy")
        val_acc_at_epoch = logs.get("val_binary_accuracy")
        if ( train_acc_at_epoch > 0.99
                and val_acc_at_epoch >0.90 ):
            self.model.stop_training = True
            self.stop_epoch = epoch

    def on_train_end( self, logs=None ):
        if self.stop_epoch > 0:
            print("--- TRAINING STOPPED AT EPOCH: ",
                  self.stop_epoch, " AS TARGET IS ACHIEVED ---")
```

3: Data Transformation

3.1: Methods

3.1.1: Creating the Data Generator for Images and Labels

```
In [ ]: def create_data_generators(train_dir, valid_dir, test_dir):
    """
    Objective:
    To fetch the auto-labelled data from the directories containing
    the sub-directories with the same name as the labels containing
    the enhanced images.

    Args:
    train_dir   : the address of the directory containing the labelled
                  sub-directories that contain the images for training.
    valid_dir   : the address of the directory containing the labelled
                  sub-directories that contain the images for validation.
    test_dir    : the address of the directory containing the labelled
                  sub-directories that contain the images for testing.

    Returns     : the generators containing the training, validation and
                  testing images along with the labels.
    Return Type : generators
    """

    # Clearing the memory TensorFlow/Keras states:
    tf.keras.backend.clear_session()

    # Training Dataset:
    train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rescale = 1./255
    )
    train_gen = train_datagen.flow_from_directory(
        train_dir,
        batch_size = BATCH_SIZE,
        target_size = IMG_SHAPE[:2],
        class_mode = "binary"
    )

    # Validation Dataset:
    valid_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rescale = 1./255
    )
    valid_gen = valid_datagen.flow_from_directory(
      valid_dir,
      batch_size = BATCH_SIZE,
      target_size = IMG_SHAPE[:2],
      class_mode = "binary"
    )


    # Testing Dataset:
    test_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rescale = 1./255
    )
    test_gen = test_datagen.flow_from_directory(
      test_dir,
      batch_size = BATCH_SIZE,
      target_size = IMG_SHAPE[:2],
      class_mode = "binary",
    )

    return train_gen,valid_gen,test_gen
```

4: Exploratory Data Analysis (E.D.A.)

4.1: Methods for E.D.A.

4.1.1: Plotting the Distribution

```
In [ ]: def plot_distribution(category, frequency,
                      bar_title: str, pie_title: str,
                      x_label: str, y_label: str,
                      fig_size: tuple):
    """
    Objective:
        To plot the given discrete distribution in the form of
        bar and pie charts.

    Args:
        category  : an iterable containing the categories
        frequency : an iterable containing the frequency of each category
        bar_title : title of the bar plot
        pie_title : title of the pie plot
        x_label   : label for the x-axis of the bar plot
        y_label   : label for the y-axis of the bar plot

    Returns : None
    """

    plt.figure(figsize=fig_size)

    # Bar Plot
    plt.subplot(1, 2, 1, title=bar_title, xlabel=x_label, ylabel=y_label)
    sns.barplot(
        x = frequency,
        y = category,
        orient="h"
    )
    for i in range(len(category)):
        plt.annotate(s=str(frequency[i]),xy=(min(frequency),i))

    # Pie Plot
    plt.subplot(1,2,2, title=pie_title)
    plt.pie(
        labels=category, x=frequency,
        autopct="%.2f%%",textprops=None
    )
```

4.1.2: Plotting the Sample Images (by category)

```
In [ ]: def plot_sample_images(category, train_dir: str):
    """
    Objective:
        To plot the sample images from each set of categories.

    Args:
        category    : an iterable containing the categories
        train_dir   : the path of the training directory containing the
                      sub-directories. The sub-directories bear
                      the same name as the categories, as well as
                      the images belonging to that category.

    Returns : None
    """

    count = 0
    plt.figure(figsize = (15,10))
    for i in range(2):
        img_list = random.choices(
            os.listdir( os.path.join(train_dir,category[i]) ),
            k = 3
        )
        for j in range(3):
            count += 1
            plt.subplot( 2, 3, count,
                         title = str(category[i])+" Image "+str(j+1)
                       )
            img = PIL.Image.open(
                os.path.join(train_dir,category[i],img_list[j])
            )
            img = img.resize( size=IMG_SHAPE[:2] )
            plt.imshow(np.array(img), cmap="gray")

    del count
```

4.1.3: Plotting the Mean Image (by category)

$\mu_x = \mathbb{E}(x) = \frac{\sum_{r=1}^{n} x}{N} \qquad \forall \quad x \in \mathbb{R}^n$

```
In [ ]: def get_mean_img(dir: str):
    """
    Objective:
        To get the mean image from all the images in the given directory.
        This is done by taking the mean of each position in the
        image matrix across all the images present in the directory.

    Args:
        dir     : the path (dtype=str) of the directory containing all
                  the images belonging a particular category.

    Returns     : the mean image
    Return Type : numpy.ndarray
    """

    img_list = [
            PIL.Image.open(
                os.path.join(dir, filename)
            ).resize( size=IMG_SHAPE[:2] )
            for filename in os.listdir(dir)
            if filename.endswith(".jpeg")
            and np.array(
                PIL.Image.open(
                    os.path.join(dir,filename)
                    )
                ).ndim==2
        ]

    mean_img = sum(
        [img for img in map(lambda x:np.array(x)/255, img_list)]
    )/len(img_list)

    del img_list

    return mean_img
```

```
In [ ]: def plot_mean_images(category, train_dir: str):
    """
    Objective:
        To plot the Mean Image for each category present in the
        training directory in the form of sub-directories.

    Args:
        category    : a list containing the PIL Images.
        train_dir   : the path of the training directory (dtype=str)
                      containing the sub-directories belonging
                      to different categories. The sub-categories
                      contain the images belonging to that category.

    Returns : None
    """

    plt.figure( figsize = (10,5))
    plt.title("MEAN IMAGES BY CATEGORY")
    for i in range(len(category)):
        print("Plotting Mean Image for the category: ",category[i])
        plt.title("MEAN IMAGE BY CATEGORY")
        plt.subplot( 1, 2, i+1, title=category[i] )
        dir = os.path.join( train_dir, category[i] )
        mean_img = get_mean_img(dir)
        plt.imshow(mean_img, cmap="gray")
```

4.1.4: Plotting the Mean Absolute Deviation (M.A.D.) Fraction Image (by category)

$M.A.D. = \mathbb{E}(\frac{|x-\mu_x|}{\mu_x+1}) \qquad \forall \quad x \in \mathbb{R}^n$

where:

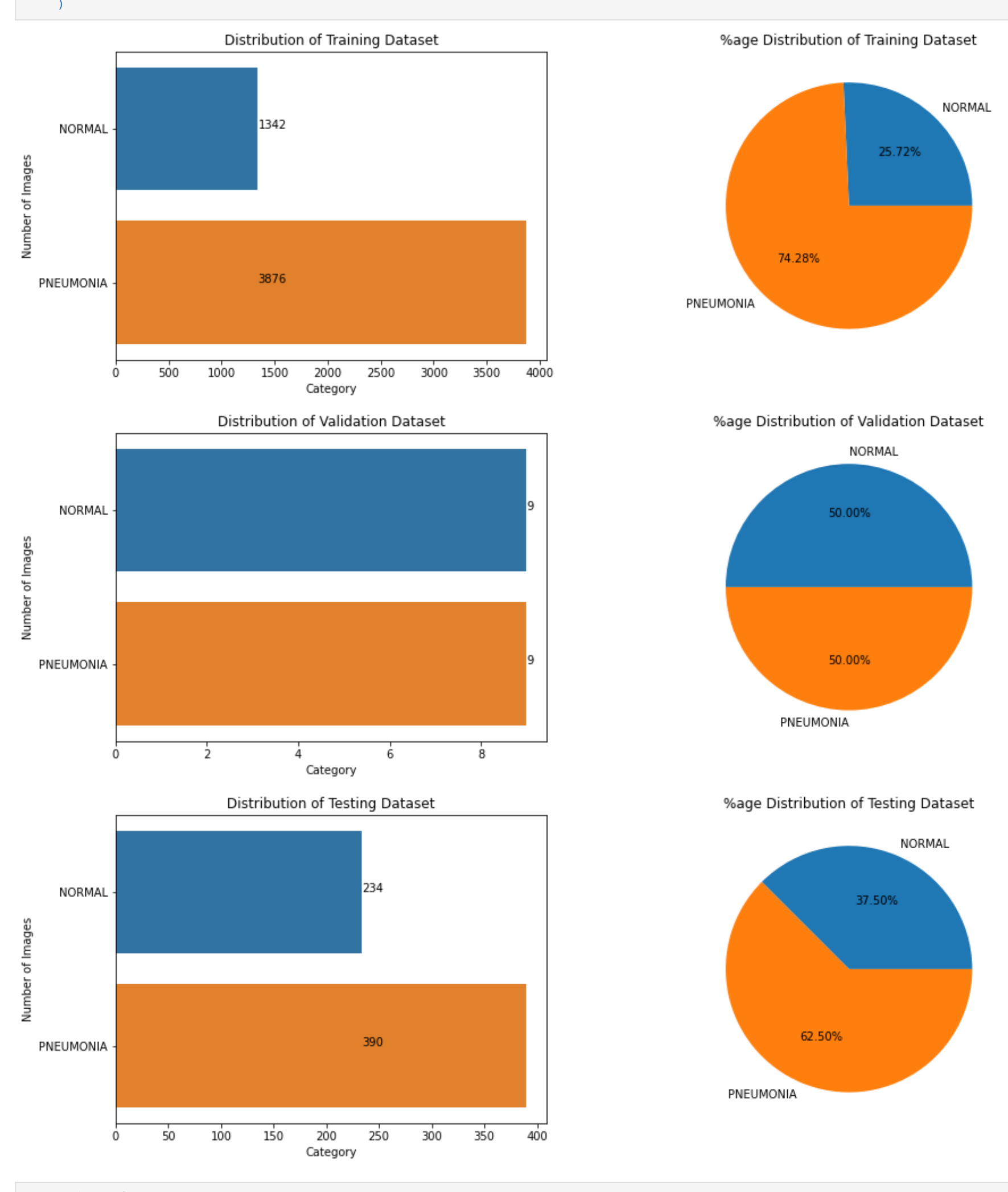
$\mu_x = \mathbb{E}(x) = \frac{\sum_{r=1}^{n} x}{N} \qquad \forall \quad x \in \mathbb{R}^n$

```
In [ ]: def get_mad_frac_img(dir):
    """
    Objective:
        To get the Mean Absolute Deviation (M.A.D.) image about the
        mean image from all the images in the given directory.
        This is done by taking the mean of each position in the
        image matrix across all the images, and calculating the
        absolute difference between the mean value and the original value
        for each position.
        This difference is then averaged to get
        the M.A.D., and is then divided by the mean value to get
        the fraction. The values in the mean image in the denominator
        have to be non-zero. Hence, 1 is added to each position in
        the mean image in the denominator.

    Args:
        dir     : the path of the directory containing all the images
                  belonging a particular category.

    Returns     : the mean image
    Return Type : numpy.ndarray
    """

    mean_img = get_mean_img(dir)

    img_list = [
            PIL.Image.open(
                os.path.join(dir, filename)
            ).resize( size=IMG_SHAPE[:2] )
            for filename in os.listdir(dir)
            if filename.endswith(".jpeg")
            and np.array(
                PIL.Image.open(
                    os.path.join(dir,filename)
                    )
                ).ndim==2
        ]
    mad_frac_img = sum(
        (np.abs(mean_img-img)/(mean_img+1)
         for img in map(lambda x:np.array(x)/255, img_list)
        )
    )/len(img_list)

    del img_list, mean_img

    return mad_frac_img
```

```
In [ ]: def plot_mad_frac_images(category, train_dir):
    """
    Objective:
        To plot the M.A.D. Fraction Image for each category present in
        the training directory in the form of sub-directories.

    Args:
        category    : a list containing the PIL Images.
        train_dir   : the path of the training directory (dtype=str)
                      containing the sub-directories belonging
                      to different categories. The sub-categories
                      contain the images belonging to that category.

    Returns : None
    """

    plt.figure(figsize=(10,5))
    plt.title("M.A.D FRACTION IMAGES BY CATEGORY")
    for i in range(len(category)):
        print("Plotting M.A.D. Fraction Image for the category: ",
              category[i])
        plt.subplot( 1, 2, i+1, title=category[i] )
        dir = os.path.join( train_dir, category[i] )
        mad_frac_img = get_mad_frac_img(dir)
        sns.heatmap(mad_frac_img)
```

4.2: Distribution of the Dataset

```
In [ ]: ls1 = [ "Training", "Validation", "Testing" ]
ls2 = [ TRAIN_DIR, VALID_DIR, TEST_DIR ]
for i in range(len(ls1)):
    plot_distribution(
        category = [ "NORMAL","PNEUMONIA" ],
        frequency = [
            len(os.listdir(os.path.join(ls2[i],x)))
            for x in [ "NORMAL","PNEUMONIA" ]
        ],
        bar_title = "Distribution of "+ls1[i]+" Dataset",
        pie_title = "%age Distribution of "+ls1[i]+" Dataset",
        x_label="Category", y_label="Number of Images",
        fig_size=(15,5)
    )
```




```
In [ ]: del ls1, ls2
```

4.3: Image Sizes

```
In [ ]: size = []
for i in range(2):
    dir = os.path.join(TRAIN_DIR, ["NORMAL","PNEUMONIA"][i])
    for filename in os.listdir(dir):
        if filename.endswith(".jpeg"):
            img = np.array(
                PIL.Image.open(os.path.join(dir, filename))
            )
            if img.shape not in size:
                size.append(img.shape)

print("Size of various images in the dataset:")
for i in size:
    print(i)
```

```
In [ ]: print("Number of unique image sizes: ",len(size))
```

```
Number of unique image sizes:  4366
```

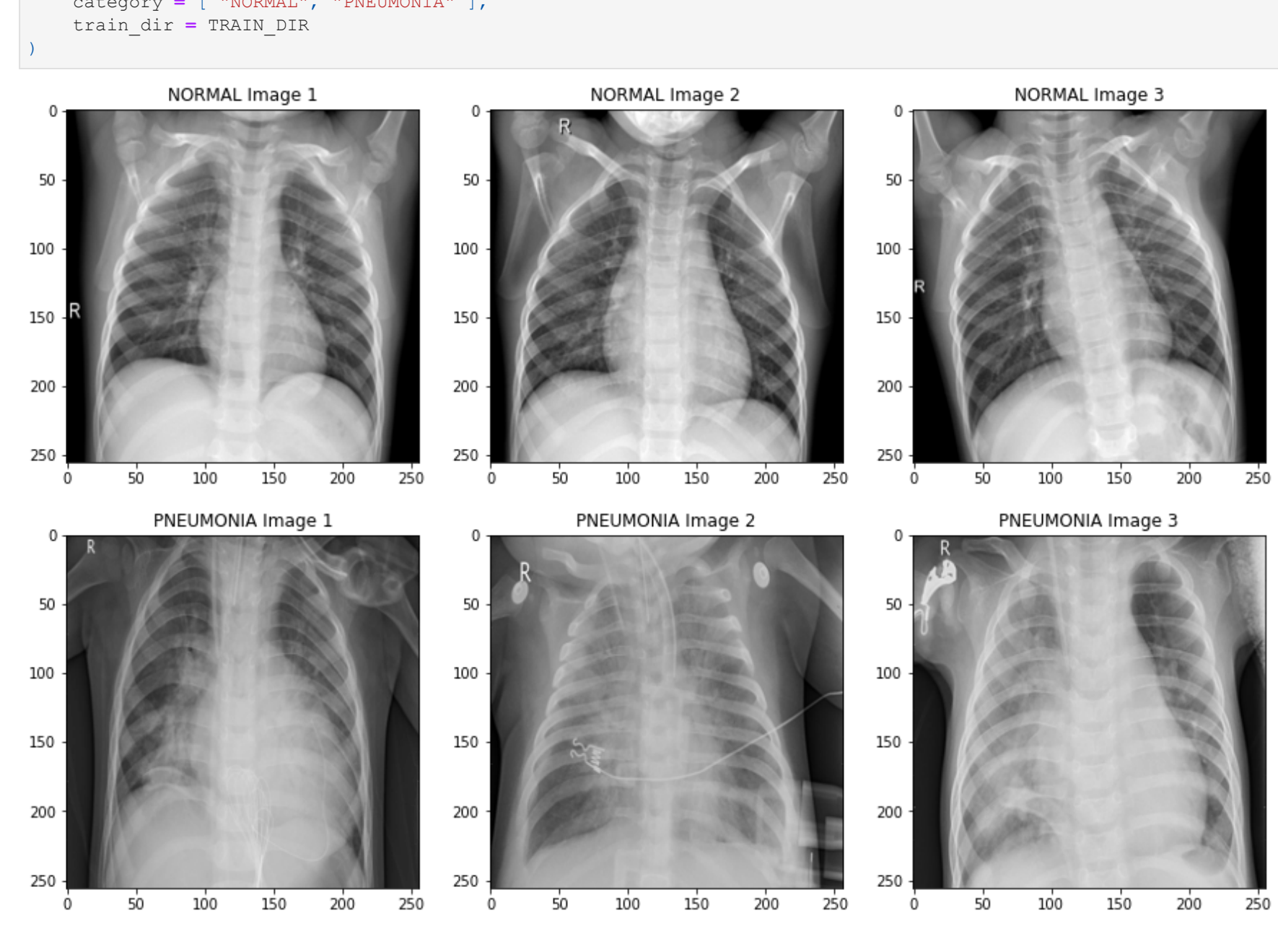
```
In [ ]: print("Min Width: ",min(map(lambda x:x[0], size)))
print("Max Width: ",max(map(lambda x:x[0], size)))
print("Min Height: ",min(map(lambda x:x[1], size)))
print("Max Height: ",max(map(lambda x:x[1], size)))
print("Avg Width: ",sum(map(lambda x:x[0], size))/len(size))
print("Avg Height: ",sum(map(lambda x:x[1], size))/len(size))
```

```
Min Width:  127
Max Width:  2663
Min Height:  384
Max Height:  2916
Avg Width:  1005.840357306459
Avg Height:  1354.620705451214
```

```
In [ ]: del size
```

4.4: Sample Images by Category

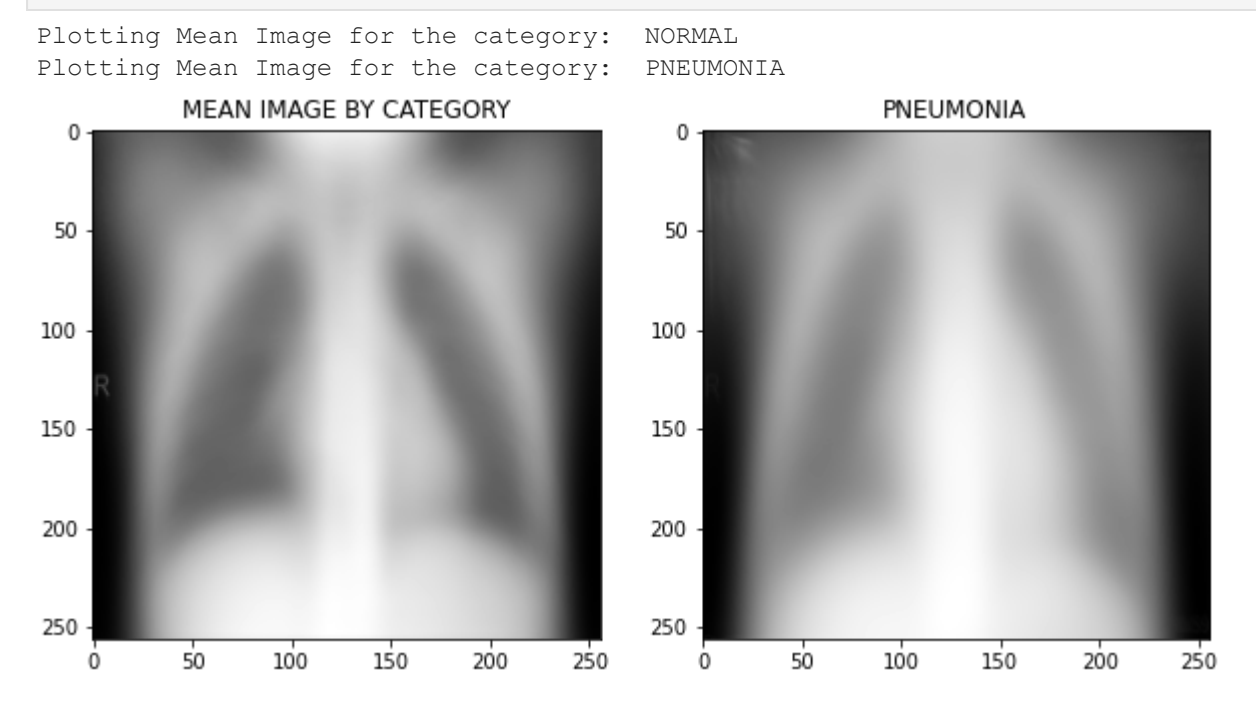
```
In [ ]: plot_sample_images(
    category = [ "NORMAL", "PNEUMONIA" ],
    train_dir = TRAIN_DIR
)
```



4.5: Mean Image by Category

```
In [ ]: plot_mean_images(
    category = [ "NORMAL", "PNEUMONIA" ],
    train_dir = TRAIN_DIR
)
```

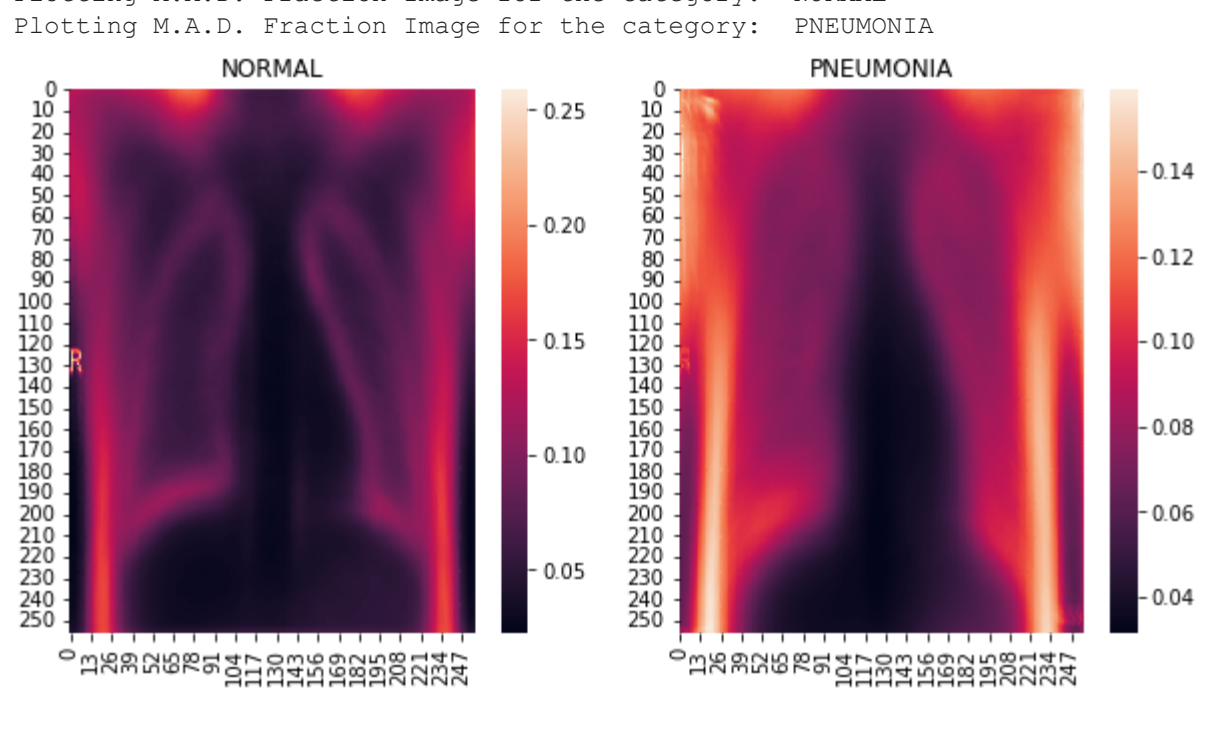
```
Plotting Mean Image for the category:  NORMAL
Plotting Mean Image for the category:  PNEUMONIA
```



4.6: M.A.D. Fraction Image by Category

```
In [ ]: plot_mad_frac_images(
    category = [ "NORMAL", "PNEUMONIA" ],
    train_dir = TRAIN_DIR
)
```

```
Plotting M.A.D. Fraction Image for the category:  NORMAL
Plotting M.A.D. Fraction Image for the category:  PNEUMONIA
```



5: Deep Learning

5.1: Check Model

5.1.1: Fetching the Dataset

```
In [ ]: TRAIN_GEN,VALID_GEN, TEST_GEN = create_data_generators(
            TRAIN_DIR, VALID_DIR, TEST_DIR
        )
```

```
Found 5216 images belonging to 2 classes.
Found 16 images belonging to 2 classes.
Found 624 images belonging to 2 classes.
```

```
In [ ]: print("CLASS INDICES")
        print("Training: ", TRAIN_GEN.class_indices)
        print("Validation: ", VALID_GEN.class_indices)
        print("Testing: ", TEST_GEN.class_indices)
```

```
CLASS INDICES
Training:  {'NORMAL': 0, 'PNEUMONIA': 1}
Validation:  {'NORMAL': 0, 'PNEUMONIA': 1}
Testing:  {'NORMAL': 0, 'PNEUMONIA': 1}
```

5.1.2: Defining the Check Model

```
In [ ]: check_model = tf.keras.Sequential(
            [
                tf.keras.layers.Conv2D(
                    filters = 128,
                    kernel_size = (3,3),
                    activation="relu",
                    input_shape=IMG_SHAPE
                ),
                tf.keras.layers.MaxPool2D(
                    pool_size = (2,2)
                ),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(
                    units = 64,
                    activation="relu"
                ),
                tf.keras.layers.Dense(
                    units = 1,
                    activation="sigmoid"
                )
            ]
        )
```

```
In [ ]: check_model.compile(
            optimizer = tf.keras.optimizers.Adam(),
            loss = tf.keras.losses.BinaryCrossentropy(),
            metrics = METRICS
        )
```

5.1.3: Training the Check Model

```
In [ ]: ! rm -rf {TB_CHECK_LOGS}
```

```
In [ ]: check_history = check_model.fit(
            TRAIN_GEN,
            validation_data = VALID_GEN,
            epochs = 5,
            callbacks = [
                tf.keras.callbacks.TensorBoard(
                    log_dir = TB_CHECK_LOGS
                )
            ]
        )
```

```
Epoch 1/5
652/652 [==============================] - 88s 117ms/step - loss: 2.8539 - binary_accuracy: 0.8637 - true_posit
ives: 3489.0000 - true_negatives: 1016.0000 - false_positives: 325.0000 - false_negatives: 386.0000 - auc: 0.90
88 - val_loss: 185.7148 - val_binary_accuracy: 0.6250 - val_true_positives: 3.0000 - val_true_negatives: 7.0000
- val_false_positives: 1.0000 - val_false_negatives: 5.0000 - val_auc: 0.5703
Epoch 2/5
652/652 [==============================] - 81s 124ms/step - loss: 1.1284 - binary_accuracy: 0.9176 - true_posit
ives: 3564.0000 - true_negatives: 1222.0000 - false_positives: 119.0000 - false_negatives: 311.0000 - auc: 0.94
51 - val_loss: 0.4826 - val_binary_accuracy: 0.8750 - val_true_positives: 7.0000 - val_true_negatives: 7.0000 -
val_false_positives: 1.0000 - val_false_negatives: 1.0000 - val_auc: 0.9688
Epoch 3/5
652/652 [==============================] - 81s 125ms/step - loss: 0.1986 - binary_accuracy: 0.9467 - true_posit
ives: 3677.0000 - true_negatives: 1261.0000 - false_positives: 80.0000 - false_negatives: 198.0000 - auc: 0.969
5 - val_loss: 1.1917 - val_binary_accuracy: 0.8750 - val_true_positives: 8.0000 - val_true_negatives: 6.0000 -
val_false_positives: 2.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9375
Epoch 4/5
652/652 [==============================] - 78s 120ms/step - loss: 0.2028 - binary_accuracy: 0.9492 - true_posit
ives: 3685.0000 - true_negatives: 1266.0000 - false_positives: 75.0000 - false_negatives: 190.0000 - auc: 0.971
7 - val_loss: 1.1953 - val_binary_accuracy: 0.8750 - val_true_positives: 8.0000 - val_true_negatives: 6.0000 -
val_false_positives: 2.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9375
Epoch 5/5
652/652 [==============================] - 82s 126ms/step - loss: 0.1410 - binary_accuracy: 0.9617 - true_posit
ives: 3726.0000 - true_negatives: 1290.0000 - false_positives: 51.0000 - false_negatives: 149.0000 - auc: 0.980
3 - val_loss: 0.1194 - val_binary_accuracy: 1.0000 - val_true_positives: 8.0000 - val_true_negatives: 8.0000 -
val_false_positives: 0.0000e+00 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
```

5.1.4: Evaluating the Performance of the Check Model

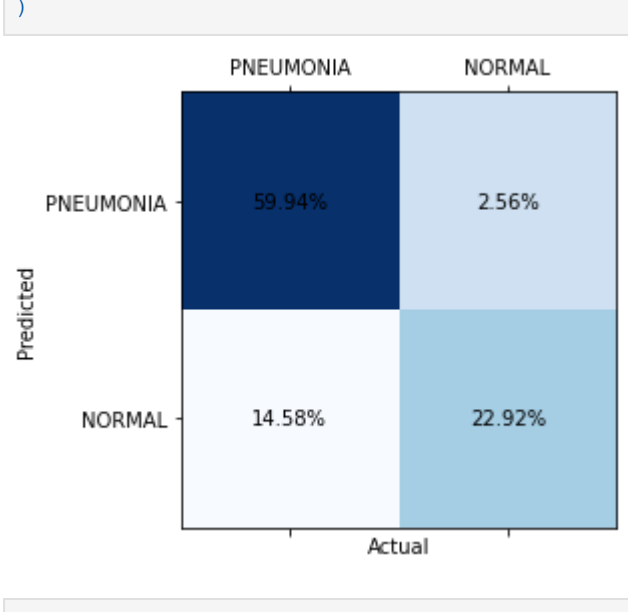
```
In [ ]: eval_check_dict = check_model.evaluate(
            TEST_GEN,
            batch_size = BATCH_SIZE,
            return_dict = True
        )
        print("EVALUATION OF THE CHECK MODEL")
        print(eval_check_dict)
```

```
78/78 [==============================] - 7s 92ms/step - loss: 1.8959 - binary_accuracy: 0.8285 - true_positive
s: 374.0000 - true_negatives: 143.0000 - false_positives: 91.0000 - false_negatives: 16.0000 - auc: 0.8403
EVALUATION OF THE CHECK MODEL
{'loss': 1.8958603143692017, 'binary_accuracy': 0.8285256624221802, 'true_positives': 374.0, 'true_negatives':
143.0, 'false_positives': 91.0, 'false_negatives': 16.0, 'auc': 0.8402585983276367}
```

5.1.5: Visualizing the Check Model's Performance

```
In [ ]: %reload_ext tensorboard
        %tensorboard --logdir {TB_CHECK_LOGS} --port=6006
```

```
In [ ]: plot_confusion_matrix(
            TP = eval_check_dict["true_positives"],
            TN = eval_check_dict["true_negatives"],
            FP = eval_check_dict["false_positives"],
            FN = eval_check_dict["false_negatives"]
        )
```




```
In [ ]: display_metrics(
            TP = eval_check_dict["true_positives"],
            TN = eval_check_dict["true_negatives"],
            FP = eval_check_dict["false_positives"],
            FN = eval_check_dict["false_negatives"]
        )
```

```
METRICS OF EVALUATION
                          Value
Metric
True Positives       374.000000
True Negatives       143.000000
False Positives       91.000000
False Negatives       16.000000
Binary Accuracy        0.828526
Precision              0.804301
Recall/Sensitivity     0.958974
Specificity            0.611111
F-Score                0.874854
```

5.1.6: Saving the Check Model

```
In [ ]: ! rm "Check_Model.h5"
```

```
rm: cannot remove 'Check_Model.h5': No such file or directory
```

```
In [ ]: check_model.save(
            "Check_Model.h5"
        )
```

5.2: Hyperparameter Tuning

5.2.1: Fetching the Dataset

```
In [ ]: TRAIN_GEN,VALID_GEN, TEST_GEN = create_data_generators(
            TRAIN_DIR, VALID_DIR, TEST_DIR
        )
```

```
Found 5216 images belonging to 2 classes.
Found 16 images belonging to 2 classes.
Found 624 images belonging to 2 classes.
```

```
In [ ]: print("CLASS INDICES")
        print("Training: ", TRAIN_GEN.class_indices)
        print("Validation: ", VALID_GEN.class_indices)
        print("Testing: ", TEST_GEN.class_indices)
```

```
CLASS INDICES
Training:  {'NORMAL': 0, 'PNEUMONIA': 1}
Validation:  {'NORMAL': 0, 'PNEUMONIA': 1}
Testing:  {'NORMAL': 0, 'PNEUMONIA': 1}
```

5.2.2: Template of the Tuning Model

```
In [ ]: def build_model(hp):
            """
            Objective:
                To build the Neural Network model.

            Args:
                hp      : an instance of the Hyperparameter class

            Returns     : the model
            Return Type : tf.keras.Sequential
            """

            # Clearing the memory TensorFlow/Keras states:
            tf.keras.backend.clear_session()

            model = tf.keras.Sequential()

            # Convolutional Block 1:
            model.add(
                tf.keras.layers.Conv2D(
                    filters = hp.Choice(
                        "Filters_1",
                        values = [ 4, 8, 16 ]
                    ),
                    kernel_size = (3,3),
                    input_shape = IMG_SHAPE,
                    activation = "relu"
                )
            )
            model.add(
                tf.keras.layers.MaxPool2D(
                    pool_size = (2,2)
                )
            )

            # Convolutional Block 2:
            model.add(
                tf.keras.layers.Conv2D(
                    filters = hp.Choice(
                        "Filters_2",
                        values = [ 16, 32, 64 ]
                    ),
                    kernel_size = (3,3),
                    activation = "relu"
                )
            )
            model.add(
                tf.keras.layers.MaxPool2D(
                    pool_size = (2,2)
                )
            )

            # Intermediate Layers:
            model.add(
                tf.keras.layers.BatchNormalization()
            )
            model.add(
                tf.keras.layers.Flatten()
            )

            # Dense Block:
            model.add(
                tf.keras.layers.Dense(
                    units = hp.Int(
                        "Units_1",
                        min_value = 32,
                        max_value = 128,
                        step = 8
                    ),
                    activation = hp.Choice(
                        "Activation_1",
                        values = [ "relu", "sigmoid", "tanh" ]
                    )
                )
            )
            model.add(
                tf.keras.layers.Dropout(
                    rate = hp.Float(
                        "Dropout_Rate_1",
                        min_value = 0.10,
                        max_value = 0.25
                    )
                )
            )
            model.add(
                tf.keras.layers.Dense(
                    units = hp.Int(
                        "Units_2",
                        min_value = 8,
                        max_value = 32,
                        step = 3
                    ),
                    activation = hp.Choice(
                        "Activation_2",
                        values = [ "relu", "sigmoid", "tanh" ]
                    )
                )
            )
            model.add(
                tf.keras.layers.Dense(
                    units = 1,
                    activation = "sigmoid"
                )
            )

            # Compiling the model:
            model.compile(
                optimizer = tf.keras.optimizers.Adam(
                    learning_rate = hp.Float(
                        "Learning_Rate",
                        min_value = 1e-6,
                        max_value = 1e-2,
                        sampling = "log"
                    )
                ),
                loss = tf.keras.losses.BinaryCrossentropy(),
                metrics = METRICS
            )

            return model
```

5.2.3: Tuning using Bayesian Optimization

```
In [ ]: ! rm -rf {TB_TUNE_LOGS}
```

```
In [ ]: tuner = kt.BayesianOptimization(
            hypermodel = build_model,
            objective = kt.Objective("binary_accuracy","max"),
            max_trials = 15,
            directory = "HPTuning_BayesOpt"
        )
```

```
In [ ]: hp_history = tuner.search(
            TRAIN_GEN,
            validation_data = VALID_GEN,
            batch_size = BATCH_SIZE,
            epochs = 5,
            shuffle = True,
            callbacks = [
                EarlyStopAtStagnancy(patience=2),
                tf.keras.callbacks.TensorBoard(
                    log_dir = TB_TUNE_LOGS
                )
            ]
        )
```

```
Trial 15 Complete [00h 05m 35s]
binary_accuracy: 0.9762269854545593

Best binary_accuracy So Far: 0.9942484498023987
Total elapsed time: 01h 10m 46s
```

```
In [ ]: tuner.search_space_summary()
```

```
Search space summary
Default search space size: 8
Filters_1 (Choice)
{'default': 4, 'conditions': [], 'values': [4, 8, 16], 'ordered': True}
Filters_2 (Choice)
{'default': 16, 'conditions': [], 'values': [16, 32, 64], 'ordered': True}
Units_1 (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 128, 'step': 8, 'sampling': None}
Activation_1 (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'sigmoid', 'tanh'], 'ordered': False}
Dropout_Rate_1 (Float)
{'default': 0.1, 'conditions': [], 'min_value': 0.1, 'max_value': 0.25, 'step': None, 'sampling': None}
Units_2 (Int)
{'default': None, 'conditions': [], 'min_value': 8, 'max_value': 32, 'step': 3, 'sampling': None}
Activation_2 (Choice)
{'default': 'relu', 'conditions': [], 'values': ['relu', 'sigmoid', 'tanh'], 'ordered': False}
Learning_Rate (Float)
{'default': 1e-06, 'conditions': [], 'min_value': 1e-06, 'max_value': 0.01, 'step': None, 'sampling': 'log'}
```

5.2.4: Obtained Hyperparameters

```
In [ ]: hps = tuner.get_best_hyperparameters()[0].values
        pd.DataFrame({
            "Hyperparameter": hps.keys(),
            "Optimal Value": hps.values()
        })
```

Out[ ]:

	Hyperparameter	Optimal Value
0	Filters_1	16
1	Filters_2	32
2	Units_1	88
3	Activation_1	relu
4	Dropout_Rate_1	0.233503
5	Units_2	14
6	Activation_2	relu
7	Learning_Rate	0.000118

5.2.5: Visualizing Tuning Model's Performance

```
In [ ]: %reload_ext tensorboard
        %tensorboard --logdir {TB_TUNE_LOGS} --port=7007
```

5.3: Final Model

5.3.1: Fetching the Dataset

```
In [ ]: TRAIN_GEN,VALID_GEN, TEST_GEN = create_data_generators(
            TRAIN_DIR, VALID_DIR, TEST_DIR
        )
```

```
Found 5216 images belonging to 2 classes.
Found 16 images belonging to 2 classes.
Found 624 images belonging to 2 classes.
```

```
In [ ]: print("CLASS INDICES")
        print("Training: ", TRAIN_GEN.class_indices)
        print("Validation: ", VALID_GEN.class_indices)
        print("Testing: ", TEST_GEN.class_indices)
```

```
CLASS INDICES
Training:  {'NORMAL': 0, 'PNEUMONIA': 1}
Validation:  {'NORMAL': 0, 'PNEUMONIA': 1}
Testing:  {'NORMAL': 0, 'PNEUMONIA': 1}
```

5.3.2: Defining the Final Model

```
In [ ]: final_model = tf.keras.Sequential(
            layers = [
                tf.keras.layers.Conv2D(
                    filters = 16,
                    kernel_size = (3,3),
                    activation="relu",
                    input_shape=IMG_SHAPE
                ),
                tf.keras.layers.MaxPool2D((2,2)),
                tf.keras.layers.Conv2D(
                    filters = 32,
                    kernel_size = (3,3),
                    activation = "relu"
                ),
                tf.keras.layers.MaxPool2D((2,2)),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(
                    units = 88,
                    activation = "relu"
                ),
                tf.keras.layers.Dropout(0.233503),
                tf.keras.layers.Dense(
                    units = 14,
                    activation = "relu"
                ),
                tf.keras.layers.Dense(1, activation="sigmoid")
            ]
        )
```

```
In [ ]: # Binding the model with Algorithmic parameters:
        final_model.compile(
            optimizer = tf.keras.optimizers.Adam(
                learning_rate = 0.000118
            ),
            loss = tf.keras.losses.BinaryCrossentropy(),
            metrics = METRICS
        )
```

```
In [ ]: final_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 16)      448

 max_pooling2d (MaxPooling2D  (None, 127, 127, 16)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 32)      4640

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 32)       0
 2D)

 batch_normalization (BatchN  (None, 62, 62, 32)       128
 ormalization)

 flatten (Flatten)           (None, 123008)            0

 dense (Dense)               (None, 88)                10824792

 dropout (Dropout)           (None, 88)                0

 dense_1 (Dense)             (None, 14)                1246

 dense_2 (Dense)             (None, 1)                 15

=================================================================
Total params: 10,831,269
Trainable params: 10,831,205
Non-trainable params: 64
_________________________________________________________________
```

5.3.3: Training the Final Model

```
In [ ]: ! rm -rf {TB_FINAL_LOGS}
```

```
In [ ]: history = final_model.fit(
            TRAIN_GEN,
            validation_data = VALID_GEN,
            epochs = 50,
            callbacks = [
                tf.keras.callbacks.TensorBoard(
                    log_dir = TB_FINAL_LOGS
                ),
                EarlyStopAtBestMetrics(),
                EarlyStopAtStagnancy(patience=5)
            ]
        )
```

```
Epoch 1/50
652/652 [==============================] - 84s 127ms/step - loss: 0.1533 - binary_accuracy: 0.9429 - true_posit
ives: 3734.0000 - true_negatives: 1199.0000 - false_positives: 150.0000 - false_negatives: 149.0000 - auc: 0.97
98 - val_loss: 0.2598 - val_binary_accuracy: 0.9375 - val_true_positives: 8.0000 - val_true_negatives: 7.0000 -
val_false_positives: 1.0000 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 2/50
652/652 [==============================] - 58s 89ms/step - loss: 0.0739 - binary_accuracy: 0.9734 - true_positi
ves: 3807.0000 - true_negatives: 1270.0000 - false_positives: 71.0000 - false_negatives: 68.0000 - auc: 0.9940
- val_loss: 0.0891 - val_binary_accuracy: 0.9375 - val_true_positives: 8.0000 - val_true_negatives: 7.0000 - va
l_false_positives: 1.0000 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 3/50
652/652 [==============================] - 57s 88ms/step - loss: 0.0606 - binary_accuracy: 0.9804 - true_positi
ves: 3823.0000 - true_negatives: 1291.0000 - false_positives: 50.0000 - false_negatives: 52.0000 - auc: 0.9951
- val_loss: 0.9067 - val_binary_accuracy: 0.7500 - val_true_positives: 8.0000 - val_true_negatives: 4.0000 - va
l_false_positives: 4.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9375
Epoch 4/50
652/652 [==============================] - 56s 86ms/step - loss: 0.0343 - binary_accuracy: 0.9870 - true_positi
ves: 3841.0000 - true_negatives: 1307.0000 - false_positives: 34.0000 - false_negatives: 34.0000 - auc: 0.9983
- val_loss: 0.0752 - val_binary_accuracy: 0.9375 - val_true_positives: 7.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 1.0000 - val_auc: 1.0000
Epoch 5/50
652/652 [==============================] - 56s 85ms/step - loss: 0.0319 - binary_accuracy: 0.9883 - true_positi
ves: 3841.0000 - true_negatives: 1314.0000 - false_positives: 27.0000 - false_negatives: 34.0000 - auc: 0.9988
- val_loss: 0.0057 - val_binary_accuracy: 1.0000 - val_true_positives: 8.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 6/50
652/652 [==============================] - 60s 92ms/step - loss: 0.0258 - binary_accuracy: 0.9893 - true_positi
ves: 3847.0000 - true_negatives: 1313.0000 - false_positives: 28.0000 - false_negatives: 28.0000 - auc: 0.9994
- val_loss: 0.9808 - val_binary_accuracy: 0.6875 - val_true_positives: 8.0000 - val_true_negatives: 3.0000 - va
l_false_positives: 5.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9922
Epoch 7/50
652/652 [==============================] - 56s 86ms/step - loss: 0.0163 - binary_accuracy: 0.9941 - true_positi
ves: 3858.0000 - true_negatives: 1327.0000 - false_positives: 14.0000 - false_negatives: 17.0000 - auc: 0.9998
- val_loss: 0.0056 - val_binary_accuracy: 1.0000 - val_true_positives: 8.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
--- TRAINING STOPPED AT EPOCH:  6  AS TARGET IS ACHIEVED ---
```

5.3.4: Evaluating the Performance of the Final Model

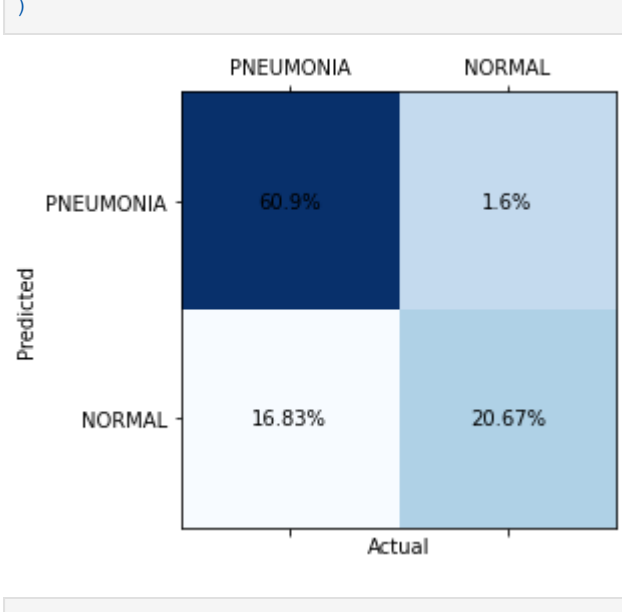
```
In [ ]: eval_final_dict = final_model.evaluate(
            TEST_GEN,
            batch_size = BATCH_SIZE,
            return_dict = True
        )
        print("EVALUATION OF THE FINAL MODEL")
        print(eval_final_dict)
```

```
78/78 [==============================] - 8s 108ms/step - loss: 1.0048 - binary_accuracy: 0.8157 - true_positive
s: 380.0000 - true_negatives: 129.0000 - false_positives: 105.0000 - false_negatives: 10.0000 - auc: 0.8756
EVALUATION OF THE FINAL MODEL
{'loss': 1.0047998428344727, 'binary_accuracy': 0.8157051205635071, 'true_positives': 380.0, 'true_negatives':
129.0, 'false_positives': 105.0, 'false_negatives': 10.0, 'auc': 0.8755807280540466}
```

5.3.5: Visualizing the Final Model's Performance

```
In [ ]: %reload_ext tensorboard
        %tensorboard --logdir {TB_FINAL_LOGS} --port=8008
```

```
In [ ]: plot_confusion_matrix(
            TP = eval_final_dict["true_positives"],
            TN = eval_final_dict["true_negatives"],
            FP = eval_final_dict["false_positives"],
            FN = eval_final_dict["false_negatives"]
        )
```




```
In [ ]: display_metrics(
            TP = eval_final_dict["true_positives"],
            TN = eval_final_dict["true_negatives"],
            FP = eval_final_dict["false_positives"],
            FN = eval_final_dict["false_negatives"]
        )
```

```
METRICS OF EVALUATION
                          Value
Metric
True Positives       380.000000
True Negatives       129.000000
False Positives      105.000000
False Negatives       10.000000
Binary Accuracy        0.815705
Precision              0.783505
Recall/Sensitivity     0.974359
Specificity            0.551282
F-Score                0.868571
```

5.3.6: Saving the Final Model

```
In [ ]: final_model.save(
            "Pneumonia_Classifier_Model.h5"
        )
```

6: Downloading the Data

```
In [ ]: ls = [
            "HPTuning_BayesOpt",
            "TB_Logs",
            "Check_Model.h5",
            "Pneumonia_Classifier_Model.h5"
        ]
```

```
In [ ]: with zipfile.ZipFile('Pneumonia_Classifier.zip','w') as zip:
            for obj in ls:
                zip.write(obj, compress_type=zipfile.ZIP_DEFLATED)
```

5. Result Analysis

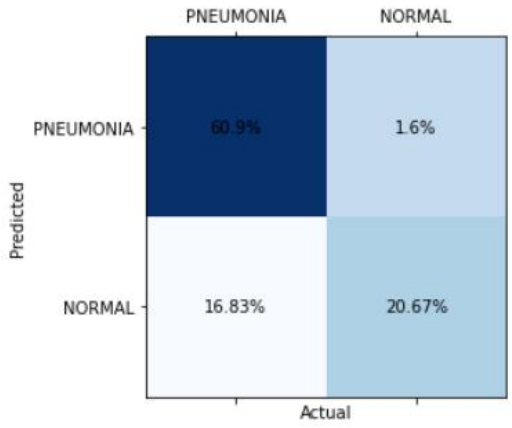
5.1: Results

5.1.1: Training Set

```
Epoch 1/50
652/652 [==============================] - 84s 127ms/step - loss: 0.1533 - binary_accuracy: 0.9429 - true_posit
ives: 3734.0000 - true_negatives: 1199.0000 - false_positives: 150.0000 - false_negatives: 149.0000 - auc: 0.97
98 - val_loss: 0.2598 - val_binary_accuracy: 0.9375 - val_true_positives: 8.0000 - val_true_negatives: 7.0000 -
val_false_positives: 1.0000 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 2/50
652/652 [==============================] - 58s 89ms/step - loss: 0.0739 - binary_accuracy: 0.9734 - true_positi
ves: 3807.0000 - true_negatives: 1270.0000 - false_positives: 71.0000 - false_negatives: 68.0000 - auc: 0.9940
- val_loss: 0.0891 - val_binary_accuracy: 0.9375 - val_true_positives: 8.0000 - val_true_negatives: 7.0000 - va
l_false_positives: 1.0000 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 3/50
652/652 [==============================] - 57s 88ms/step - loss: 0.0606 - binary_accuracy: 0.9804 - true_positi
ves: 3823.0000 - true_negatives: 1291.0000 - false_positives: 50.0000 - false_negatives: 52.0000 - auc: 0.9951
- val_loss: 0.9067 - val_binary_accuracy: 0.7500 - val_true_positives: 8.0000 - val_true_negatives: 4.0000 - va
l_false_positives: 4.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9375
Epoch 4/50
652/652 [==============================] - 56s 86ms/step - loss: 0.0343 - binary_accuracy: 0.9870 - true_positi
ves: 3841.0000 - true_negatives: 1307.0000 - false_positives: 34.0000 - false_negatives: 34.0000 - auc: 0.9983
- val_loss: 0.0752 - val_binary_accuracy: 0.9375 - val_true_positives: 7.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 1.0000 - val_auc: 1.0000
Epoch 5/50
652/652 [==============================] - 56s 85ms/step - loss: 0.0319 - binary_accuracy: 0.9883 - true_positi
ves: 3841.0000 - true_negatives: 1314.0000 - false_positives: 27.0000 - false_negatives: 34.0000 - auc: 0.9988
- val_loss: 0.0057 - val_binary_accuracy: 1.0000 - val_true_positives: 8.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
Epoch 6/50
652/652 [==============================] - 60s 92ms/step - loss: 0.0258 - binary_accuracy: 0.9893 - true_positi
ves: 3847.0000 - true_negatives: 1313.0000 - false_positives: 28.0000 - false_negatives: 28.0000 - auc: 0.9994
- val_loss: 0.9808 - val_binary_accuracy: 0.6875 - val_true_positives: 8.0000 - val_true_negatives: 3.0000 - va
l_false_positives: 5.0000 - val_false_negatives: 0.0000e+00 - val_auc: 0.9922
Epoch 7/50
652/652 [==============================] - 56s 86ms/step - loss: 0.0163 - binary_accuracy: 0.9941 - true_positi
ves: 3858.0000 - true_negatives: 1327.0000 - false_positives: 14.0000 - false_negatives: 17.0000 - auc: 0.9998
- val_loss: 0.0056 - val_binary_accuracy: 1.0000 - val_true_positives: 8.0000 - val_true_negatives: 8.0000 - va
l_false_positives: 0.0000e+00 - val_false_negatives: 0.0000e+00 - val_auc: 1.0000
--- TRAINING STOPPED AT EPOCH:  6  AS TARGET IS ACHIEVED ---
```

5.1.2: Test Set

```
78/78 [==============================] - 8s 108ms/step - loss: 1.0048 - binary_accuracy: 0.8157 - true_positive
s: 380.0000 - true_negatives: 129.0000 - false_positives: 105.0000 - false_negatives: 10.0000 - auc: 0.8756
EVALUATION OF THE FINAL MODEL
{'loss': 1.0047998428344727, 'binary_accuracy': 0.8157051205635071, 'true_positives': 380.0, 'true_negatives':
129.0, 'false_positives': 105.0, 'false_negatives': 10.0, 'auc': 0.8755807280540466}
```



METRICS OF EVALUATION

Metric	Value
True Positives	380.000000
True Negatives	129.000000
False Positives	105.000000
False Negatives	10.000000
Binary Accuracy	0.815705
Precision	0.783505
Recall/Sensitivity	0.974359
Specificity	0.551282
F-Score	0.868571

5.2: Test

Test No.	Metric	Target Value	Achieved Value	Target Value (Test Set)	Achieved Value (Test Set)	STATUS
1	**Binary Accuracy**	0.950	0.9941	0.800	0.815705	PASSED
2	**Precision**	0.900	0.9964	0.750	0.783505	PASSED
3	**Recall**	0.950	0.9956	0.900	0.974359	PASSED
4	**F-Score**	0.925	0.9960	0.820	0.868571	PASSED
5	**AUC-ROC Score**	0.850	0.9998	0.800	0.875581	PASSED

Reference: Section 2 - Problem Statement

6. Conclusion

Since the objectives of the project as stated in Section 2 - Problem Statement have been achieved satisfactorily, we can conclude it as successful. While doing this project, a few practical lessons were also learnt, such as:

- The time taken to tune the hyper-parameters and train the model using CPU is immense. The GPU was available only for limited time, so many computations had to be done using the CPU.
- Not being able to use TPU due to non-availability, as well as not having access to Google Cloud Storage (GCS) should have been considered at the earliest stage. The TPUs are lot faster than CPU, but they require the data to be present in a GCS Bucket in the same region as the TPU.
- It would be much easier to *deploy* the model directly on cloud for using it in applications, rather than doing so on the local machine.
- The model size can be really big, The HDF5 of even a simple model can be in GigaBytes. This makes it difficult to store and deploy it using the local machine.
- It is always preferable to use classical ML models instead of DL model because of less number of trainable parameters in classical ML models. DL should be the last resort to solve a problem, the way it was in this case.
- Doing Deep Learning on Google Cloud would be lot better than doing it on the local machine:
  - Google Cloud has its own ML/DL platform specifically designed for the purpose: **Vertex AI**.
  - Vertex AI is integrable with other Google Cloud Services, such as GCS.
  - Vertex AI also has built-in Hyper-parameter Tuning Support using Bayesian Optimization.
  - The factors such as scalability and computational support are much better handled by cloud services.
- Data pre-processing/modelling and miscellaneous planning takes much more time. Once done *meticulously*, the actual implementation does not take as much time (subject to the condition that the planning was *meticulous*).
- Never undermine the power of Exploratory Data Analysis (EDA). It is capable of providing insights that might drastically reduce the problem complexity.

This project can be extended using the Google Cloud Platform (GCP) to develop it into an end-to-end full stack Deep Learning project. Only if the access to the platform was available, this too could have been achieved.

7. References

1. Dataset: Chest X-Ray Pneumonia:
https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia

2. Mayo Clinic > Pneumonia > Symptoms and Causes:
https://www.mayoclinic.org/diseases-conditions/pneumonia/symptoms-causes/syc-20354204

3. Ques: 353, List of Questions for ORAL ANSWERS [dated 05.04.2022], Rajya Sabha:
https://cms.rajyasabha.nic.in/UploadedFiles/Questions/QuestionsList/256/English_202245_SQ05.pdf

4. PIB Press Release [ID: 1845081 dated 26.07.2022]:
https://pib.gov.in/PressReleseDetailm.aspx?PRID=1845081

5. PIB Press Release [ID: 1813656 dated 05.04.2022]:
https://pib.gov.in/PressReleseDetailm.aspx?PRID=1813656

6. Analytics Vidhya > How to load Kaggle dataset directly into Google Colab:
https://www.analyticsvidhya.com/blog/2021/06/how-to-load-kaggle-datasets-directly-into-google-colab/

7. Keras-Tuner (Official Documentation):
https://keras.io/guides/keras_tuner/getting_started/
https://www.tensorflow.org/tutorials/keras/keras_tuner/

8. Analysis and Optimization of Convolutional Neural Network Architectures - Martin Thoma:
https://arxiv.org/pdf/1707.09725.pdf

9. Machine Learning Mastery > Improve Deep Learning Performance:
https://machinelearningmastery.com/improve-deep-learning-performance/

10. Gopal, M. (2019). *Applied Machine Learning*. McGraw-Hill Education.

11. Data Science - Stack Exchange:
https://datascience.stackexchange.com/

12. Stack Overflow:
https://stackoverflow.com/

13. Towards Data Science:
https://towardsdatascience.com/

14. Srinivas, N., Krause, A., Kakade, S. M., & Seeger, M. W. (2012). Information-theoretic regret bounds for gaussian process optimization in the bandit setting. *IEEE Transactions on Information Theory*, *58*(5), 3250–3265. https://doi.org/10.1109/tit.2011.2182033